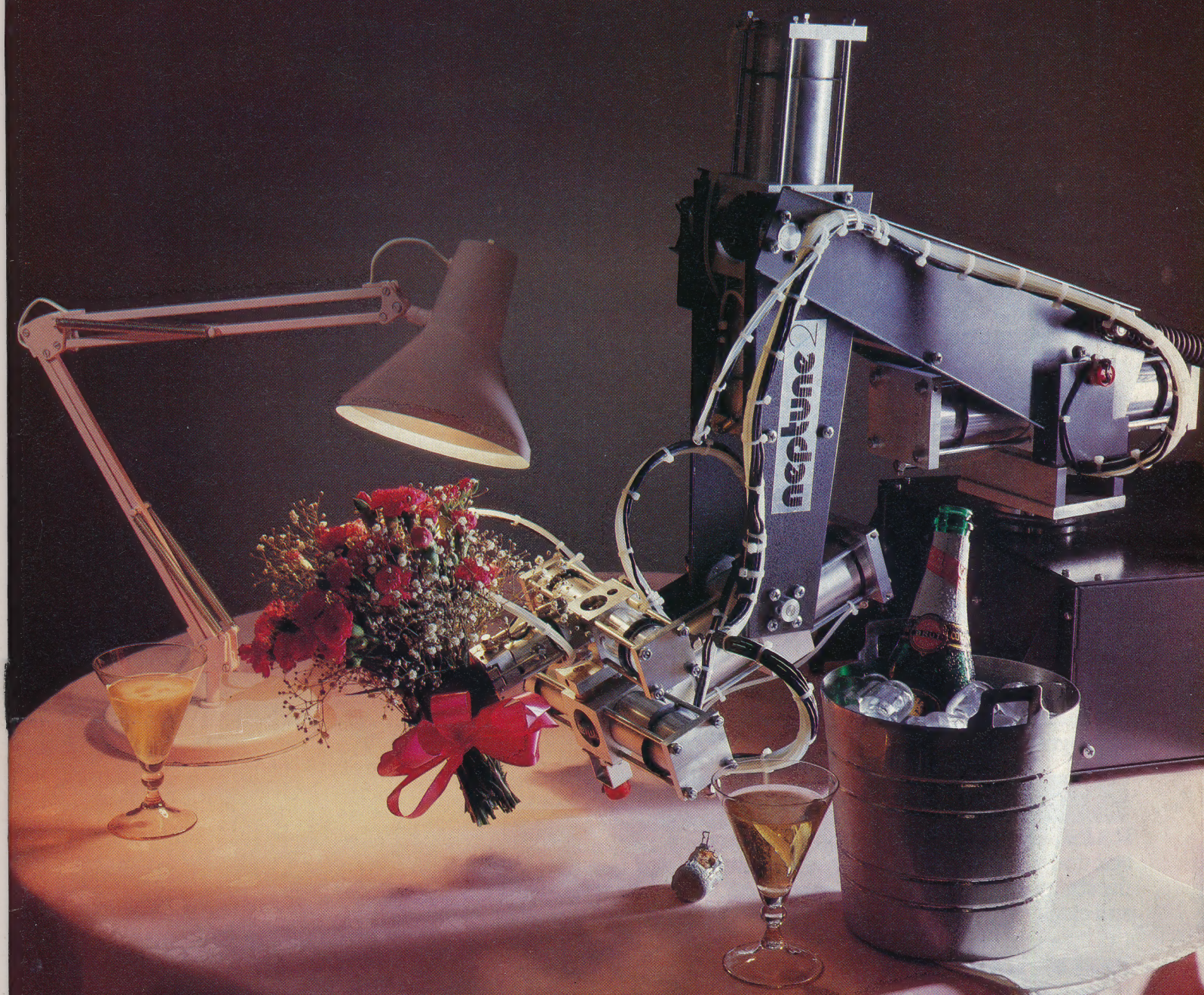
ISSN 0265-2919

80p 34

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

■ **Our hardware feature takes a comprehensive look at the most popular home micros on the market. We give prices, facilities and reviews.**

■ **We begin a new series on using spreadsheet programs on micros for household applications.**

■ **The BASIC programming course continues with another toolkit utility project – finding and altering variable names throughout a program.**

QUIZ

1) What is the purpose of the keywords displayed at the bottom of an MSX machine's power-up screen display?
2) What is a UART, and what does it do?
3) How do Cartesian co-ordinates get their name?
4) Which game, popular on mainframe computers, was the model for Atari's 'Star Raiders'?

Answers To Last Week's Quiz

1) Round brackets are used to print text and variables on the same line.
2) Postfix notation, also known as Reverse Polish, is a system of arithmetic used by digital devices in which the operators are grouped together after the operands.
3) Although the M10 is sold under the Olivetti trademark , the computer is manufactured by Kyocera of Japan.
4) A keyboard macro is a series of keystrokes programmed into the computer to perform a single operation.

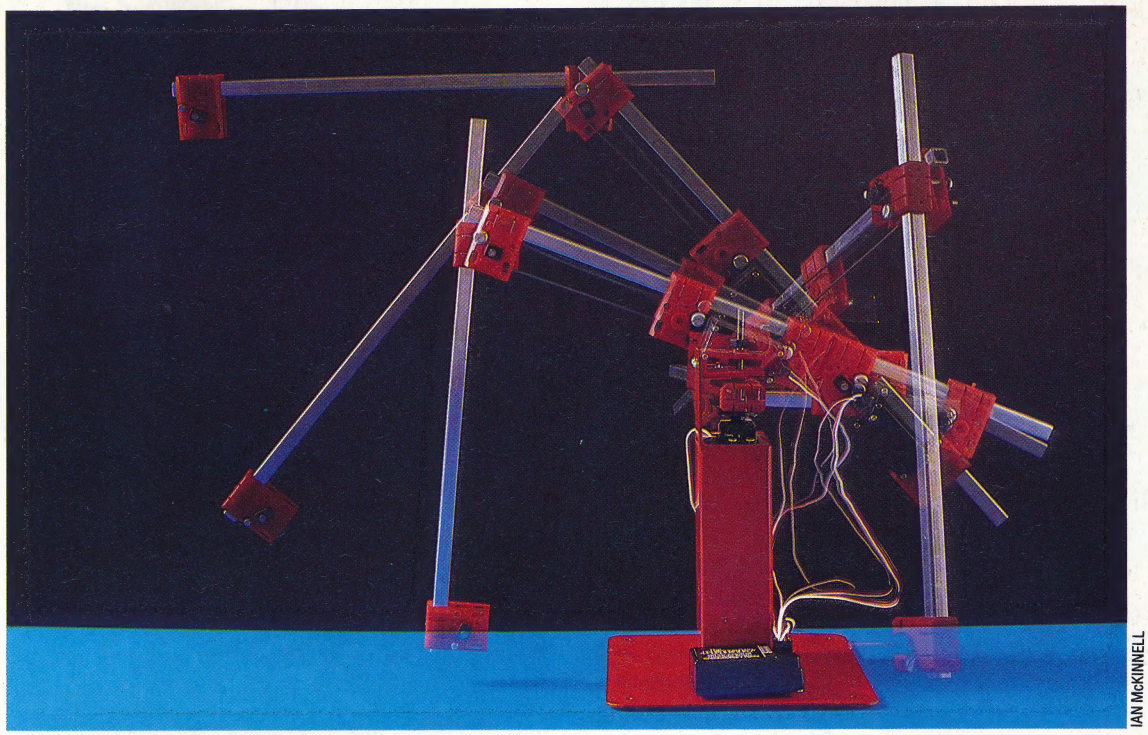
Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Sub Editor** Steve Mann; **Researcher** Melanie Davis; **Staff Writer** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellor, Mike Curtis, Steve Colwill, Chris Naylor, Max Phillips, Steve Malone, Ted Ball; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984; Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Price: 80p/IR£1. Subscription: 6 months: £23.92. 1 Year: £47.84. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 80p. Subscription: 6 months air: £37.96. Surface: £31.46. 1 year air: £75.92. Surface: £62.92. Binder: £5.00. Airmail: £5.00. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 80p. Subscription: 6 months air: £43.94. Surface: £31.46. 1 year air: £87.88. Surface: £62.92. Binder: £5.00. Airmail: £8.31. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/80p. Subscription: 6 months air: £51.74. Surface: £31.46. 1 year air: £103.48. Surface: £62.92. Binder: £5.00. Airmail: £9.44. **SOUTH AFRICA** – Price: SA R1.95. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 80p. Subscription: 6 months air: £55.38. Surface: £31.46. 1 year air £110.76. Surface: £62.92. Binder: £5.00. Airmail: £9.84. **AUSTRALIA** – Price: Aus$1.95. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZ$2.25. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 6711. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH

ROBOT ARM COURTESY OF CYBERNETIC APPLICATIONS LTD

PUMPING IRON



IAN McKINNELL

Double Jointed
For the next decade, the archetypal robot will be the simple arm equipped with a variety of 'hands' for industrial, household and hobbyist use. Very few applications really require the self-propelled autonomous thinking machine of sci-fi myth, but a programmable semi-intelligent gripper is as significant a device as the plough or the telescope

Having considered the various methods that are used to control robot movement, we now turn our attention to an equally important aspect of robot design — the control and movement of the robot's 'arm' and 'hand'.

A robot's effectiveness depends to a large degree on the precision with which it can manipulate objects. Many robots are used primarily for 'pick up and place' operations — moving components in a factory from one conveyor belt to another, for example. Thus, the design of the robot arm is of paramount importance.

In general, there are three requirements that must be considered. A system must be developed to describe the position of the arm at any time; the arm must have a 'skeleton'; and there must be a 'muscle' system that will actuate the arm and enable it to be controlled. The different ways in which these vital elements interact tend to dictate the overall appearance of robot arms. However, different types of arm may be roughly classified by considering the spatial methods used to describe the arm's exact position at any given time.

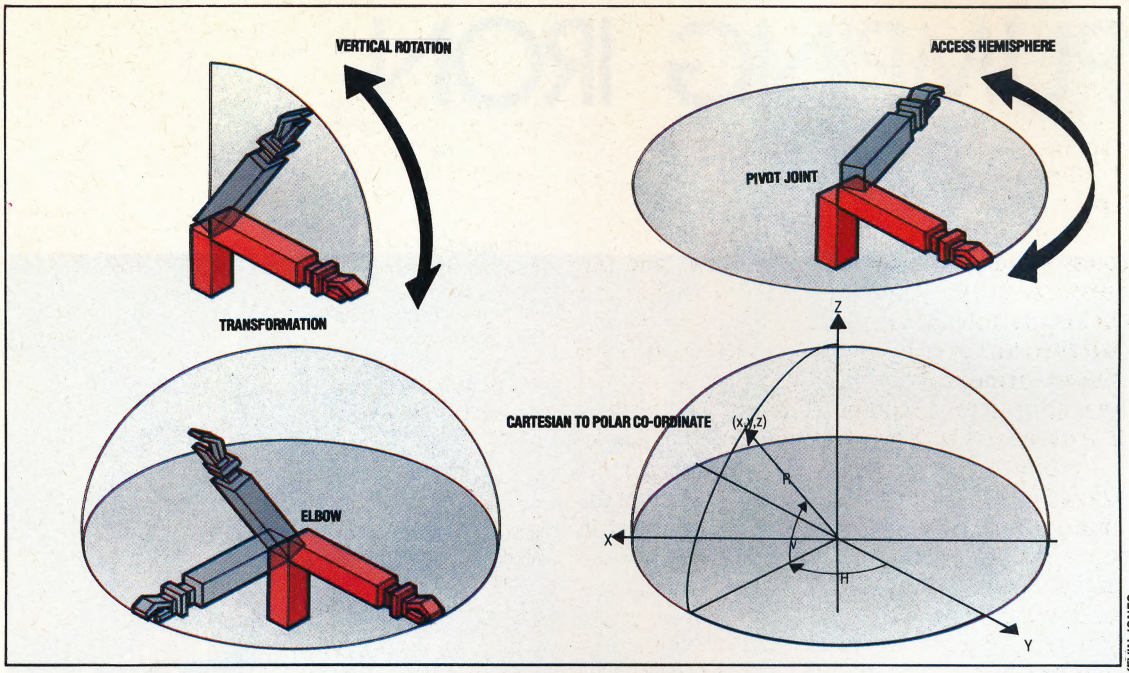
In our discussion of robot movement (see page 621), we described the Cartesian co-ordinate system. Using this method, the position of the robot on the floor was specified by means of two axes — x and y — at right angles to each other. The same principle can be applied to a robot arm, but, because an arm may move freely in three dimensions, we need to add another variable — z — to describe the arm's vertical position. Using these x, y and z co-ordinates, we can describe the position of the arm anywhere in space ('space' simply being the mathematical way of describing any open area).

It is possible to construct a robot arm that moves exactly along these three co-ordinates: the result will be something that looks a little like an overhead gantry crane that can move up/down, side-to-side, and forwards/backwards (or all three directions in combination). Arms like this are well suited to jobs in which work is done over some fixed area. For instance, the robot might have a workbench at which all its tasks are carried out, and, in this case, a Cartesian arm will be more than adequate. But this method does have its disadvantages. For example, such arms require a substantial frame, which makes them inflexible in applications away from the workbench.

Another method of describing the position of an arm uses cylindrical co-ordinates. To understand how these work, think of an empty tin can; you will realise that any position inside the can may be described by specifying its distance from the centre of the can (using a distance variable, r); how far around the can it is from some fixed point (using an angular variable, θ); and how far up the side of the can it is (using

Robotic Rotations
The simplest arm, consisting of a gripper and a two-axis elbow joint, is capable of precise positioning in a very large volume of space as these illustrations show.

The elbow is hinged, permitting semi-circular vertical movement, and pivoted, which allows horizontal circular movement. The arm moves to any point on the access hemisphere by rotations at the pivot and hinge. These can be derived by trigonometry from the Cartesian (x,y and z) co-ordinates of the point as shown: H, the pivot rotation, is equal to ARCTAN(x/y), while V, the hinge rotation, is ARCSIN(z/R). The arm is programmed with the Cartesian locations of objects; it transforms each set of co-ordinates into two rotations which it sends to its two servomotors, thus effecting movement



KEVIN JONES

another distance variable, z). So by using cylindrical co-ordinates it would be very easy to develop a system that could pick out any object from a specified position inside the can.

Arms that use spherical co-ordinates take this process one step further by specifying a position in terms of two angles and one distance. In this case, 'distance' is the length of the arm, and the two angles are the amount by which the base rotates and the angle of elevation of the arm. Arms such as these are very much like a gun turret, in which the length of the gun barrel may be varied. Spherical co-ordinates are usually described as r, θ, and ∅. For the robot engineer, it is simple enough to design an arm that can move in and out telescopically, possibly driven by hydraulic power.

The final, and most common, method of describing the position of an arm is by using revolute co-ordinates. This is a system that is specifically designed to control robot arms by imitating the actions of the human arm. As before, three variables are needed to specify the arm's position; this time they are all angles and could be described as θ, ∅ and γ co-ordinates. θ (theta) refers to the angle through which the base is rotated; ∅ (phi) refers to the angle of elevation of the arm; and γ (gamma) describes the angle of a second arm joint.

BUILDING UP MUSCLE

The chosen co-ordinate system will dictate the type of 'skeleton' a robot arm requires. All that is needed now is some 'muscle' to power the arm's movement. In general, there are three types of robot muscle used — electrical, hydraulic and pneumatic. Let's look at these in turn.

We have already discussed electrical power in connection with robot movement. The same electrical stepper motors may be used to power robot arms. For example, they can do so directly, by having a powerful motor at each arm joint and letting this rotate by a small amount for each joint movement, or indirectly, by means of gears, pulleys or levers.

However, a better system would involve making the robot 'muscles' work in much the same way as our own — by expanding and contracting so as to act on the skeleton of the arm directly. This is done by arranging a series of pistons to act on each arm joint. These pistons may be hydraulic (using fluid) or pneumatic (using air). For use with massive industrial robots, hydraulic power is preferred as this can provide much higher pressure (giving more force to the arm) and because fluid does not compress or expand to the same extent as air does.

This means that when a piston is moved along a cylinder by hydraulic pressure it does not 'bounce', but stops at precisely the desired point. Air, by contrast, does not allow such precise positioning. No matter which system is used, single or double action pistons may be utilised to produce motion in the arm. This type of motive power is called a *linear actuator.*

A further refinement is possible. Instead of using pistons that move backwards and forwards and then translating this movement into a rotation at the joint, a *rotary actuator* may be used. This produces direct rotation in the joints by means of pressure on a vane inside a circular housing. This is a similar process to the use of an electrical stepper motor, but the hydraulic pressure means that far more power may be exerted. Pneumatic pressure is unsuitable for this type of application.

Once the mechanics of the robot arm have been decided upon, all that is needed is a 'hand' (or *end effector*) so that, once the arm is correctly positioned, it can actually do something. Here, it is instructive to think about the way a human hand works. Consider the human wrist — if this was

encased in plaster so that it could not be moved, most tasks would be much more difficult. When operating a keyboard, for example, the wrists allow your hands to move up and down as you strike the keys — this is known as 'pitch', and without it, you would have to move the whole forearm up and down when typing.

Your wrists also move from side to side as you press the different keys — this is 'yaw', and the absence of this would entail elbow movement. Once you have finished typing, you can turn your wrists so that your hands rest, thumbs upwards, at the side of the keyboard. This is known as 'roll' and would require a complicated set of shoulder movements if wrist movement was not available.

Ideally, then, these three different movements should all be built into the robot wrist. Each of the movements — pitch, yaw and roll — can act in two directions (up/down, left/right, clockwise/anticlockwise) and each of these is called a 'degree of freedom'. So a robot that incorporates pitch, yaw and roll can be said to have six degrees of freedom. Robots are built with lesser degrees of freedom — perhaps four or five — but for each reduction in wrist movement there is a corresponding increase in the movements that must be made by other, larger parts of the arm.

THE ROBOT HAND

We must now consider the design of the hand itself. The ideal configuration would be a human-like hand at the end of a human-like arm, and some robot hands do approach this definition. The most common form of robot hand is a three-fingered gripper — consisting of two fingers plus an opposing 'thumb' — which enables the robot to grasp objects in much the same way as a human hand would.

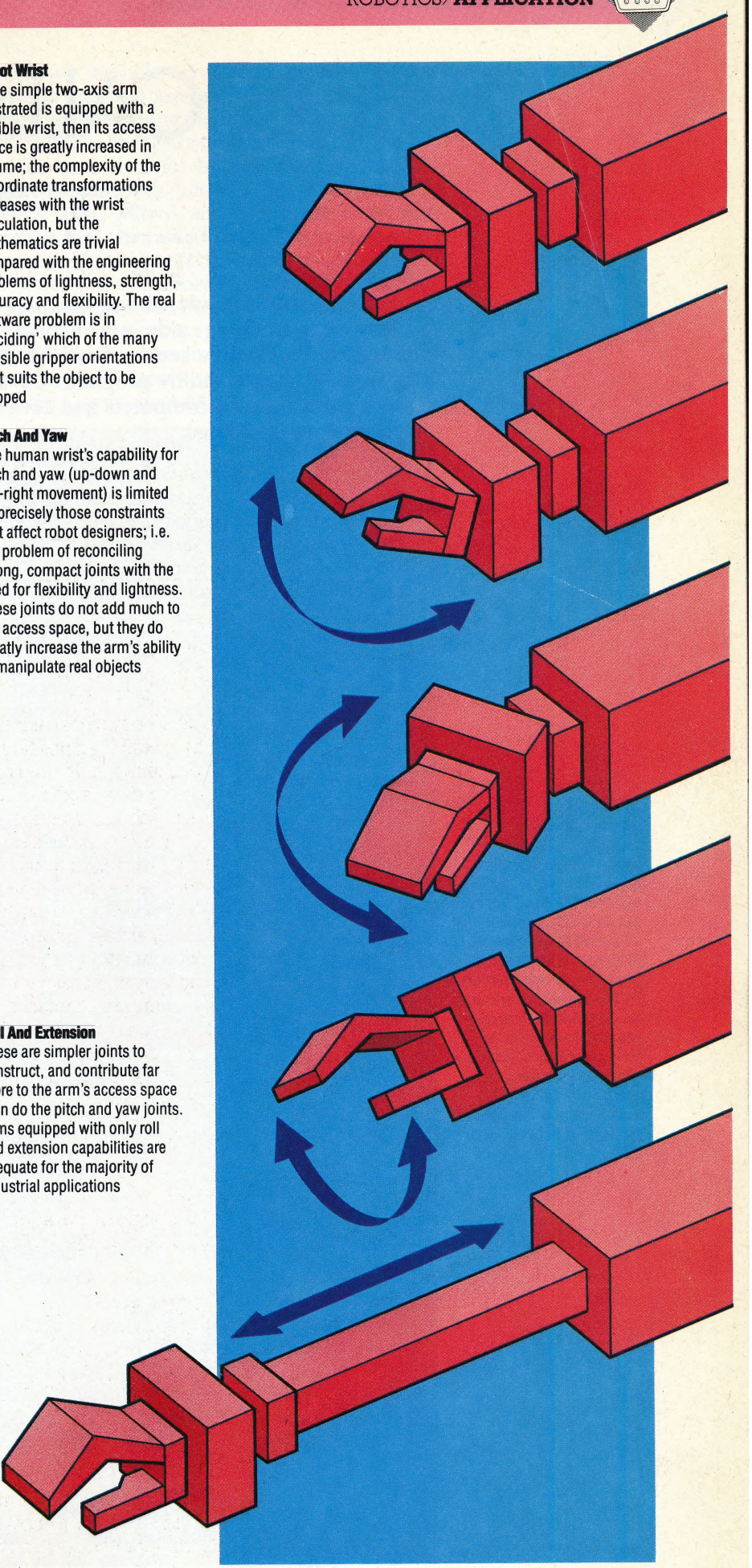
The power used to drive the hand can be any of the three types already mentioned, and will depend on the task the robot is to perform. If the hand must move large objects weighing several hundred pounds, hydraulics will probably be necessary. But for many applications, simple electrical or pneumatic power will suffice because the hand will need only to grip an object and release it when desired — if the arm and wrist have positioned the hand correctly, this will not require any great accuracy; a simple opening and closing movement will be enough.

In many cases, though, the robot arm will not be fitted with a hand. We have already used the phrase 'end effector' to describe a hand, but this can just as easily refer to many other things. A robot that is used for welding does not require a hand at all — a welding gun may be fastened directly to the wrist. In fact, some robots are capable of choosing the correct end effector for the task they are carrying out; they can discard one end effector (a screwdriver, say) and insert another (a spray gun, for example) into a standard socket at the wrist. This may not be a particularly human-like action, but it serves to make robots extremely adaptable.

Robot Wrist
If the simple two-axis arm illustrated is equipped with a flexible wrist, then its access space is greatly increased in volume; the complexity of the co-ordinate transformations increases with the wrist articulation, but the mathematics are trivial compared with the engineering problems of lightness, strength, accuracy and flexibility. The real software problem is in 'deciding' which of the many possible gripper orientations best suits the object to be gripped

Pitch And Yaw
The human wrist's capability for pitch and yaw (up-down and left-right movement) is limited by precisely those constraints that affect robot designers; i.e. the problem of reconciling strong, compact joints with the need for flexibility and lightness. These joints do not add much to the access space, but they do greatly increase the arm's ability to manipulate real objects

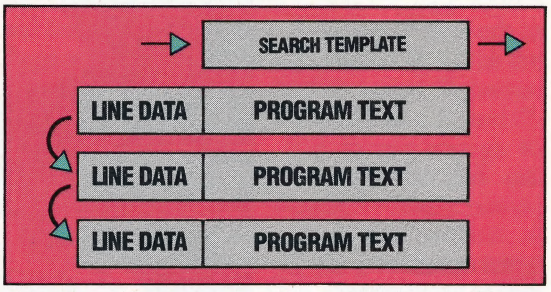
Roll And Extension
These are simpler joints to construct, and contribute far more to the arm's access space than do the pitch and yaw joints. Arms equipped with only roll and extension capabilities are adequate for the majority of industrial applications



ROLL CALL

Programs that are built in to home computers and provide editing facilities or other programming aids are known as 'utilities'. We begin a series of articles that explore a range of utility programs for the more popular home computers and develop our own BASIC utilities.

The range of utilities featured on home micros varies considerably: some computers have only a simple editor, as in the Sinclair Spectrum, while other machines feature more extensive facilities. The BBC Micro, for example, includes the TRACE and RENUMber commands: the former causes the line number of each BASIC statement to be displayed as the line is being executed, and the latter automatically renumbers the lines of a BASIC program. Both of these facilities are immensely useful in program development and debugging. But whatever is provided with your machine, it is invariably helpful to have additional utilities, and there is a wide range of commercially available programs to choose from.



Searching Hi And Lo
On the BBC Micro and the Spectrum a program line begins with three or four bytes dedicated to the line number and the length of the line. This is followed by the tokenised BASIC text. When the program encounters the line it records the line number and calculates the start address of the next line from the length of the current line. It then 'slides' the search template along the program text until it meets end-of-line, end-of-program, a REM token or a successful match

Utility programs are generally written in Assembly language, partly because of the speed of machine code and partly because it is not easy for a BASIC program to alter itself without crashing the computer. However, we will begin by looking at some simple utilities that can be writen in BASIC. In this way, we can concentrate our attention on what a utility program has to do, without having to consider other complicating details, such as the role of the computer's operating system and the BASIC interpreter.

Although it is difficult for a BASIC program to alter itself, there is no problem in creating a BASIC program that inspects another BASIC program. The utility program we give here, in versions for the Spectrum and the BBC Micro, searches through a BASIC program for the name of a variable or function, and prints the line numbers where the name is found.

Both programs begin by finding where the program text starts in the computer's memory. Then they go through the program line by line, skipping over those sections that cannot include a name and extracting all the names. The final step involves comparing each extracted name with the name the program was asked to find.

When the program starts its search of a new line of BASIC text, it first notes the line number, which in both cases is stored in two bytes, and the line's length (the number of bytes it occupies). In the BBC Micro, the line length is contained in one byte, and is the total number of bytes in the line from the line number to the end-of-line marker (ASCII code 13). In the Spectrum, the line length is stored in two bytes, and represents the number of bytes from the character following the length bytes to the end-of-line marker (thus not including the line number and line length bytes in its total).

In both versions of the program, we ignore all the REM statements and anything that is enclosed in quotes, as we will not normally have any program variables in these character strings. The BBC Micro allows you to include hexadecimal numbers in a program, prefixed by the character &. We need to make sure that our program does not mistake these hexadecimal numbers for variable names, and therefore we need to make the program skip over any strings preceded by an &. For example, we do not want our program to mistake the hex number A0 in &A0 as the variable name 'A0'.

In the Spectrum, numbers are stored in a program as the ASCII characters for the digits of the number, followed by the ASCII code byte 14, and then five bytes containing the binary equivalent of the number. Our program needs to be able to skip over the number code and the five byte binary equivalent.

Having tested for these conditions, the program proceeds to inspect the current line for any names. In both programs, a name is defined as beginning with a letter, followed by another letter or a digit. The BBC Micro version allows integer variables (distinguished by a % character after the name) and the underscore character, and both versions allow string variables, which are followed by a $ character.

The name of an array, a function or (in the BBC Micro) a procedure will be followed by an open bracket — (. Strictly speaking, this is not part of the name, but is used by the programs to distinguish these from simple variables.

There are further complications in the Spectrum program. In particular, Spectrum BASIC does not distinguish between upper- and lower-case for characters in a variable name. Thus, FRED, Fred or FRed are all treated as the same variable name. The Spectrum program, therefore, converts

all letters to upper-case before it starts its matching. The Spectrum will also allow spaces to be included in variable names, but these can lead to problems and we advise against their use.

Spectrum BASIC does not make the strict distinction found in most other BASIC dialects between string variables and string arrays — in fact, a Spectrum string variable is more like a character array. Since we can have, for example, T$ and T$(i), referring to a string and part of the same string, the program does not attempt to distinguish between simple string variables and string arrays. This is not a real limitation, however, as Spectrum BASIC does not allow you to have a string variable and array with the same name.

To use our utility program, first type it in and save it, and then merge it with the program you want searched, using the MERGE command on the Spectrum or the method described in Chapter 37 of the User Guide for the BBC Micro. Invoke the search routine with RUN 9000 (Spectrum) or GOTO 30000 (BBC Micro), and type in the name you want to find when asked for it. If you want an array name, add an open bracket at the end of the name.

Finally, as an exercise, you may like to use the principles described here to write a program that tells you which lines in a program contain a call to a specified subroutine. In the next instalment of this course we will supply a full listing of this program for the Commodore 64.

Spectrum

```
9000 INPUT "Name to search for? "; LINE T$
9010 FOR i=1 TO LEN (T$)
9020 IF T$(i)>="a" AND T$(i)<="z" THEN
LET T$(i)=CHR$ (CODE (T$(i))-32)
9030 NEXT i
9040 LET TokenforREM=234
9050 LET Quote=34
9060 LET Newline=13
9070 LET Underscore=95
9080 LET Number=14
9090 LET PROG=23635
9100 LET Textpointer=PEEK (PROG)+256*PEE
K (PROG+1)
9110 LET Lineno=256*PEEK (Textpointer)+P
EEK (Textpointer+1)
9120 IF Lineno>=9000 THEN  STOP
9130 LET Textpointer=Textpointer+2
9140 LET Textlength=PEEK (Textpointer)+2
56*PEEK (Textpointer+1)
9150 LET Textpointer=Textpointer+2
9160 LET Nextline=Textpointer+Textlength
9170 IF PEEK (Textpointer)=Newline THEN
 LET Textpointer=Textpointer+1: GO TO 91
10
9180 IF PEEK (Textpointer)<>TokenforREM
THEN  GO TO 9220
9190 REM Skip over REM line
9200  LET Textpointer=Nextline
9210 GO TO 9110
9220 IF PEEK (Textpointer)<>Quote THEN
GO TO 9280
9230 REM Skip anything between quotes, b
ut stop at end of line in case of unmatc
hed quote
9240 LET Textpointer=Textpointer+1
9250 IF PEEK (Textpointer)=Newline THEN
 LET Textpointer=Textpointer+1: GO TO 91
10
9260 IF PEEK (Textpointer)<>Quote THEN
GO TO 9240
9270 LET Textpointer=Textpointer+1
9275 GO TO 9170
9280 IF PEEK (Textpointer)<>Number THEN
 GO TO 9320
9290 REM Skip 5-byte binary number
9300 LET Textpointer=Textpointer+6
9305 GO TO 9170
9310 REM First character of name must be
 upper or lower case letter
9320 IF PEEK (Textpointer)>=CODE ("A") A
ND PEEK (Textpointer)<=CODE ("Z") THEN
LET c$=CHR$ (PEEK (Textpointer)): GO TO
9370
9330 REM Use upper case instead of lower
 case
9340 IF PEEK (Textpointer)>=CODE ("a") A
ND PEEK (Textpointer)<=CODE ("z") THEN
LET c$=CHR$ (PEEK (Textpointer)-32): GO
TO 9370
9350 LET Textpointer=Textpointer+1
9360 GO TO 9170
9370 LET n$=""
9380 LET n$=n$+c$
9390 LET Textpointer=Textpointer+1
9400 REM Letter, digit or underscore aft
er first character of name
9410 IF PEEK (Textpointer)>=CODE ("A") A
ND PEEK (Textpointer)<=CODE ("Z") THEN
LET c$=CHR$ (PEEK (Textpointer)): GO TO
9380
9420 REM Use upper case instead of lower
 case
9430 IF PEEK (Textpointer)>=CODE ("a") A
ND PEEK (Textpointer)<=CODE ("z") THEN
LET c$=CHR$ (PEEK (Textpointer)-32): GO
TO 9380
9440 IF PEEK (Textpointer)>=CODE ("0") A
ND PEEK (Textpointer)<=CODE ("9") THEN
LET c$=CHR$ (PEEK (Textpointer)): GO TO
9380
9450 IF PEEK (Textpointer)=Underscore TH
EN  GO TO 9380
9460 REM End with $ fo. string variable
9470 IF PEEK (Textpointer)=CODE ("$") TH
EN  LET n$=n$+"$": LET Textpointer=Textp
ointer+1: GO TO 9500
9480 REM ( if array or function
9490 IF PEEK (Textpointer)=CODE ("(") TH
EN  LET n$=n$+CHR$ (PEEK (Textpointer)):
 LET Textpointer=Textpointer+1
9500 IF n$=t$ THEN  PRINT n$;" IN LINE "
;Lineno
9520 GO TO 9170
```

BBC Micro

```
30000 INPUT "Name to search for";
TARGET$
30010 TokenforREM=244
30020 Quote=34
30030 Hex=38
30040 Newline=13
30050 Underscore=95
30060 Textpointer=PAGE
30070 Textpointer=Textpointer+1
30080 Lineno=256*?Textpointer+?(Textpoin
ter+1)
30090 IF Lineno>=30000 THEN END
30100 Textpointer=Textpointer+2
30110 Linelength=?Textpointer
30120 Endline=Textpointer+Linelength-3
30130 Textpointer=Textpointer+1
30140 IF ?Textpointer=Newline THEN GOTO
30070
30150 IF ?Textpointer<>TokenforREM THEN
GOTO 30180
30160    REM Skip over REM line
30170    Textpointer=Endline:GOTO 30070
30180 IF ?Textpointer<>Quote THEN GOTO 3
0240
30190    REM Skip antything between quot
es, but stop at end of line in case of u
nmatched quote
30200    Textpointer=Textpointer+1
30210    IF ?Textpointer=Newline THEN GO
TO 30070
30220    IF ?Textpointer<>Quote THEN GOT
O 30200
30230    Textpointer=Textpointer+1
30235 GOTO 30140
30240 IF ?Textpointer<>Hex THEN GOTO 303
00
30250    REM Skip hex number, to avoid c
onfusion with variable names
30260    Textpointer=Textpointer+1
30270    IF ?Textpointer>=ASC("0") AND ?
Textpointer<=ASC("9") THEN GOTO 30260
30280    IF ?Textpointer>=ASC("A") AND ?
Textpointer<=ASC("F") THEN GOTO 30260
30285    GOTO 30140
30290 REM First character of name must b
e upper or lower case letter
30300 IF ?Textpointer>=ASC("A") AND ?Tex
tpointer<=ASC("Z") THEN GOTO 30330
30310 IF ?Textpointer>=ASC("a") AND ?Tex
tpointer<=ASC("z") THEN GOTO 30330
30320 GOTO 30130
30330 Name$=""
30340 Name$=Name$+CHR$(?Textpointer)
30350 Textpointer=Textpointer+1
30360 REM Letter, digit or underscore af
ter first character
30370 IF ?Textpointer>=ASC("A") AND ?Tex
tpointer<=ASC("Z") THEN GOTO 30340
30380 IF ?Textpointer>=ASC("a") AND ?Tex
tpointer<=ASC("z") THEN GOTO 30340
30390 IF ?Textpointer>=ASC("0") AND ?Tex
tpointer<=ASC("9") THEN GOTO 30340
30400 IF ?Textpointer=Underscore THEN GO
TO 30340
30410 REM End with $ for string variable
, % for integer variable
30420 IF ?Textpointer=ASC("$") THEN Name
$=Name$+CHR$(?Textpointer):Textpointer=T
extpointer+1:GOTO 30450
30430 IF ?Textpointer=ASC("%") THEN Name
$=Name$+CHR$(?Textpointer):Textpointer=T
extpointer+1
30440 REM ( if array, procedure or funct
ion
30450 IF ?Textpointer=ASC("(") THEN Name
$=Name%+CHR$(?Textpointer):Textpointer=T
extpointer+1
30460 IF Name$=TARGET$ THEN PRINT Name$;
" IN LINE ";Lineno
30470 GOTO 30140
30480 END
```

BUGS IN THE WORKS

Having covered turtle geometry in some detail, the course moves on to look at LOGO's use of sprites. We start with a discussion of the basic principles behind turtle sprites, drawing our examples from Commodore LOGO, and show how the language can use sprites to create animation effects.

Using LOGO, sprites act in a similar way to turtles, obeying all the commands that a turtle obeys. Unlike turtles, however, we can define the shape of a sprite ourselves — although these shapes do not rotate on the screen as the sprite's heading changes, in the way that a turtle does.

In Commodore LOGO, the turtle is counted as sprite number 0, and there are seven other sprites — numbered 1 to 7. To begin with, sprite 0 is the 'current' sprite, and obeys all the sprite commands entered. To make sprite 1 the current sprite, you simply type in TELL 1. From then on, all sprite commands will be obeyed by sprite 1 until a different current sprite is specified.

After typing TELL 1, however, you won't see anything on the screen. This is because all the sprites, other than the turtle, begin as 'hidden' objects and have their pens up. In order to see sprite 1, and see where it moves, you have to type ST, and a vague square will appear on the screen. Experiment with this sprite grid, using the turtle commands FD, BK, RT, LT, PD, PU, ST, HT, and so on.

If you move sprite 1 to the same position as sprite 0 (the turtle) you will notice that it appears to be behind the turtle. In general, lower numbered sprites are shown 'in front of' higher numbered sprites. This is useful for three-dimensional effects.

There is a sprite editor on the Commodore LOGO utilities disk. Read this in by typing READ "SPRED. To edit the shape of sprite 1, first make it the current sprite with TELL 1 and then type EDSH. The display will show a much enlarged view of the sprite grid and we can now move the cursor around the screen. Pressing the asterisk key (*) will fill in a pixel, while pressing the space bar will blank it out.

Having designed your sprite, press Control-C to define the shape. If the sprite is not visible, try entering ST. This same shape can now be given to other sprites as well. SETSHAPE 1 will give the current sprite the same definition as sprite 1. Having defined a set of shapes, you can save the sprites to a file with SAVESHAPES "FILENAME, and read them back with READSHAPES "FILENAME.

There is a well-known mathematical problem in which four bugs are placed at the corners of a square. They are all set off at the same speed and each follows the bug to its right. The objective is to trace their paths. We give a LOGO program here that implements the problem using sprites.

The procedures that we give simply position a copy of the same sprite at each corner of the square, and then set them off following each other. The bug shape is defined as sprite 3, and the others are all given the same shape using SETSHAPE 3 in the position procedure.

The heart of our solution lies in the FOLLOW procedure. In this, X and Y are first set to the x and y co-ordinates of the sprite that is being followed (:B), and then the sprite that is doing the following (:A) has its heading set towards this point. To do this, we use the primitive TOWARDS. This takes two inputs, which represent the co-ordinates of the point to be headed towards, and outputs the heading from the current sprite to that point.

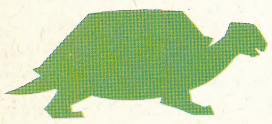
ANIMATION

One interesting use of sprite graphics is in the creation of animation effects. A series of sprite shapes representing the same object is defined. Each of these is slightly different from the one before, and when they are run together they give the effect of motion. Commodore LOGO gives three shapes that are a crude attempt at a man running. The following procedures set up the screen, and then set the three shapes in motion.

```
TO RUNN
   TELL 0
   DRAW
   PU
   BIGX BIGY
   SETH 90
   RUNNING 2
END

TO RUNNING :SHAPE
   FD 5
   SETSHAPE :SHAPE
   IF :SHAPE = 4 THEN MAKE "SHAPE 1
   RUNNING :SHAPE + 1
END
```

Before running these procedures, read in the file SPRITES from the utilities disk. This contains a number of useful procedures including BIGX and BIGY, which double the size of a sprite. SMALLX and SMALLY are the reverse procedures: these are used to return a sprite to its original size. Read in the three sprites by typing READSHAPES "RUNNER, and then run the procedures.

We also define four sprite shapes on the following page, which we will use in a game in the next instalment.

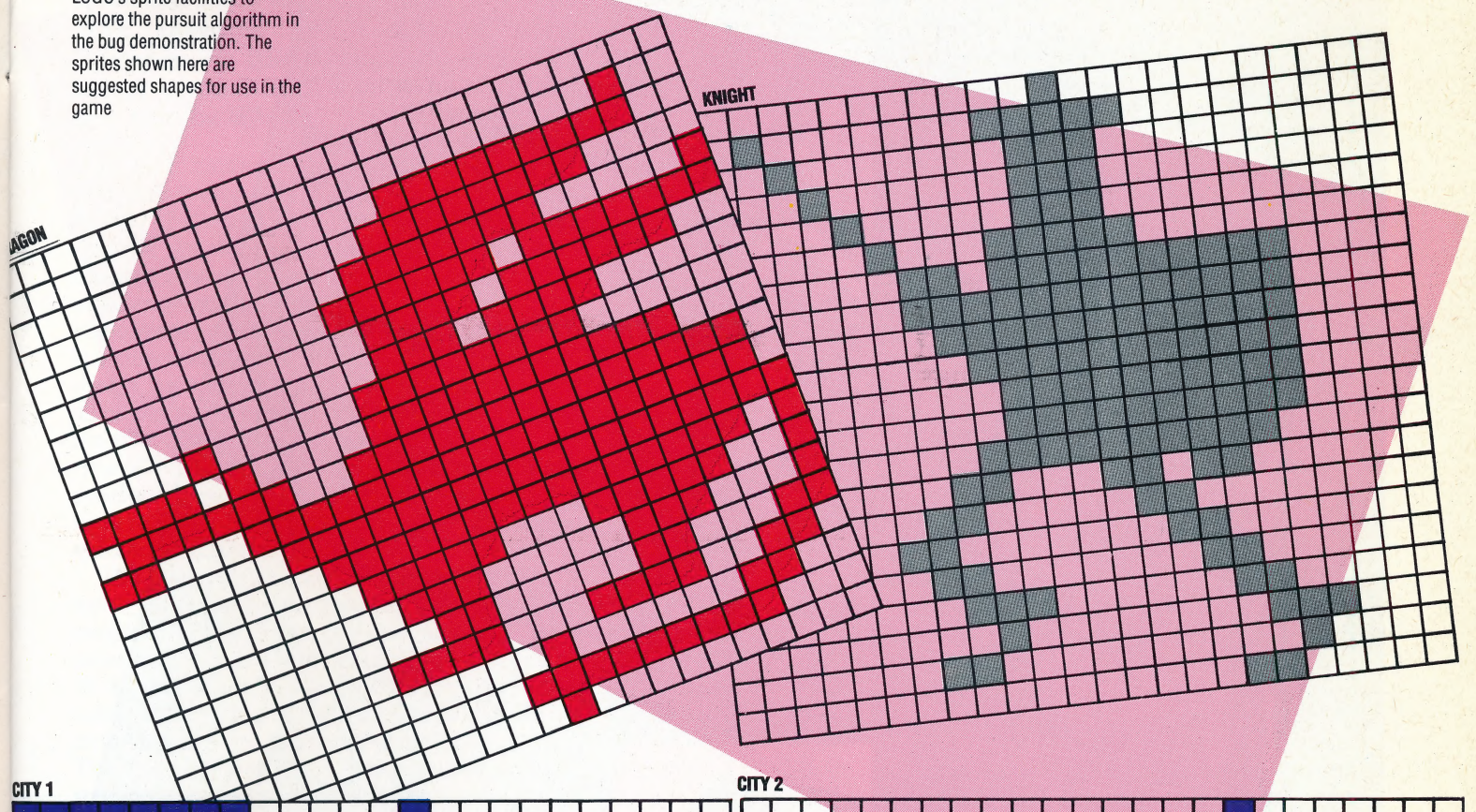


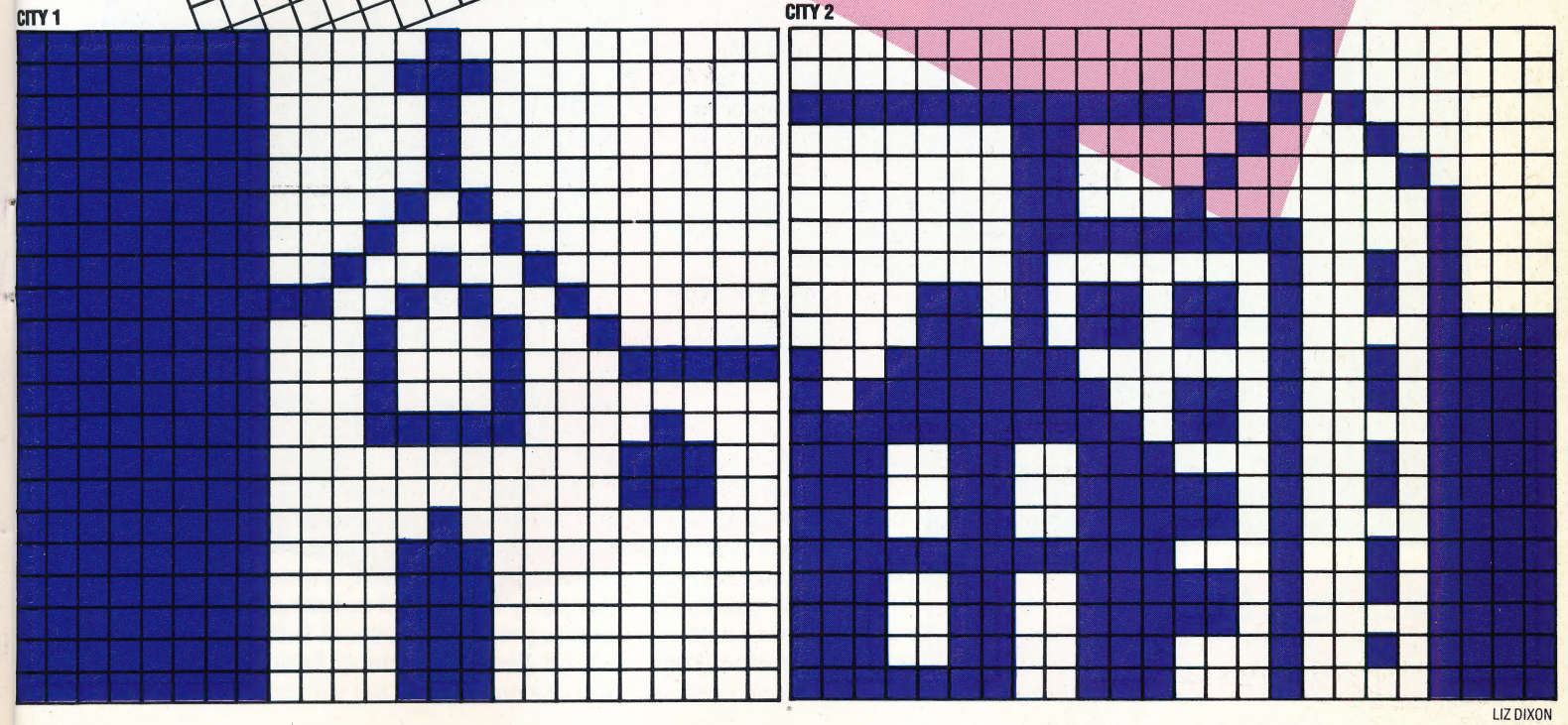
Logo Flavours

Neither Spectrum LOGO nor the Apple LOGOs features sprite graphics.
Atari users should note the following differences:
1) There are only four sprites available.
2) For SETSHAPE use SETSH.
3) The sprite editor is included among the primitives. Pressing the Space bar fills an empty pixel, or blanks a filled one.

Dragon Rampant

In the next instalment we will publish the knight and dragon game, which uses Commodore LOGO's sprite facilities to explore the pursuit algorithm in the bug demonstration. The sprites shown here are suggested shapes for use in the game



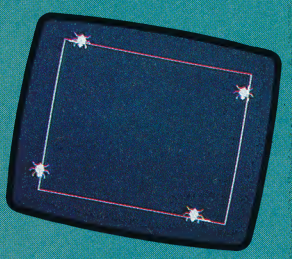


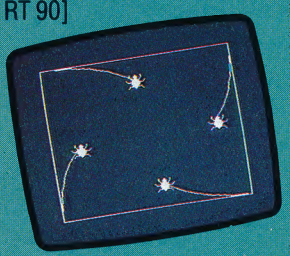
LIZ DIXON

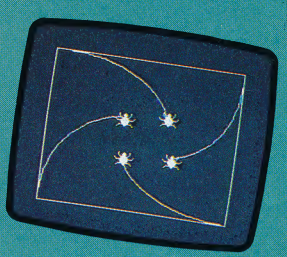
Four Bugs

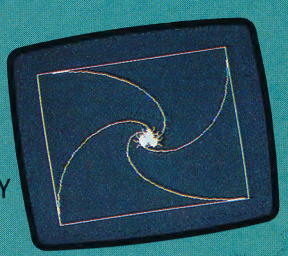
In this classic geometric demonstration each bug moves in turn directly towards the present position of the bug on its right. This algorithm inevitably produces an inward spiral, any arm of which describes the 'curve of pursuit' familiar to fighter pilots and arcade games players alike

```
TO BUGS
   SETUP
   MOVE.BUGS
END

TO SETUP
   DRAW
   FULLSCREEN
   TELL 0
   HT
   PU
   SETXY ( – 100 ) ( – 100 )
   SQUARE 200
   POSITION 1 ( – 100 ) ( – 100 )
   POSITION 2 ( – 100 ) 100
   POSITION 3 100 100
   POSITION 4 100 ( – 100 )
END

TO SQUARE :SIDE
   PD
   REPEAT 4 [FD :SIDE RT 90]
   PU
END

TO POSITION :NO :X :Y
   TELL :NO
   SETSHAPE 3
   PU
   SETXY :X :Y
   PD
   ST
END

TO MOVE.BUGS
   FOLLOW 1 2
   FOLLOW 2 3
   FOLLOW 3 4
   FOLLOW 4 1
   MOVE.BUGS
END

TO FOLLOW :A :B
   TELL :B
   MAKE "X XCOR
   MAKE "Y YCOR
   TELL :A
   SETH TOWARDS :X :Y
   FD 10
END
```









IAN McKINNELL

Three Bugs Exercise

Write a LOGO program for another bugs problem, this time with three bugs at the corners of a triangle

Lunar Lander Project

This is a program that we have developed as one solution to the project set in the last instalment (see page 655). Type LAND to play this simple version of the game:

```
TO LAND
   SETUP PLAY
END

TO SETUP
   DRAW DRAW.PLATFORM SET.ROCKET
END

TO DRAW.PLATFORM
   PU SETXY ( – 20 ) ( – 60 ) PD SETXY 20 ( – 60 )
   PU
END

TO SET.ROCKET
   SETXY 0 120 MAKE "VEL 0 MAKE "FUEL 50
END

TO PLAY
   COMMAND MOVE
   IF YCOR < – 53 THEN BOOM STOP
   GRAVITY FUEL.REPORT PLAY
END

TO COMMAND
   IF READKEY = "F THEN BURN
END

TO READKEY
   IF RC? THEN OUTPUT RC
   OUTPUT "
END

TO BURN
   IF :FUEL > 0 THEN MAKE "VEL :VEL + 0.5 MAKE
   "FUEL :FUEL – 1
END

TO MOVE
   SETY YCOR + :VEL
END

TO BOOM
   IF :VEL > ( – 1 ) THEN PRINT [YOU LANDED
   SAFELY, CONGRATULATIONS] STOP
   PRINT [THE IMPACT KILLED ALL THE CREW!]
END

TO GRAVITY
   MAKE "VEL :VEL – 0.2
END

TO FUEL.REPORT
   ( PRINT "FUEL :FUEL )
END
```

EASTERN PROMISE

Over a dozen Japanese companies, many of them household names in other areas of consumer electronics, have agreed on the MSX blueprint for a standard home computer. We take a look at the first MSX micros to reach the UK – the Sony Hit-Bit and Toshiba HX-10.

The MSX standard (see page 141) dictates the CPU that is used (Z80); the minimum amount of ROM (32 Kbytes) and RAM (eight Kbytes); the type of graphics and sound chip; keyboard contents (though layout can vary); the minimum number of interfaces and their design; graphic and text screens and, of course, the BASIC language that is contained in ROM. Because MSX is a standard design, it is to be expected that MSX machines will all be similar. Manufacturers have flexibility in the amount of memory beyond the minimum, the type of keyboard used, and the number of extra interfaces. In practice, Sony and Toshiba, like most MSX manufacturers, have gone for a higher specification than the minimum requirement.

The Sony Hit-Bit and the Toshiba HX-10 both have good quality keyboards, although some people might find the keys too sensitive. Both micros come with 64 Kbytes of main memory, and 16 Kbytes of additional RAM dedicated to the video display. This gives a total of 80 Kbytes — more than is provided on most home computers. The Sony and Toshiba models each have a standard Centronics printer interface and a pair of joystick sockets — items that are often optional extras on home computers.

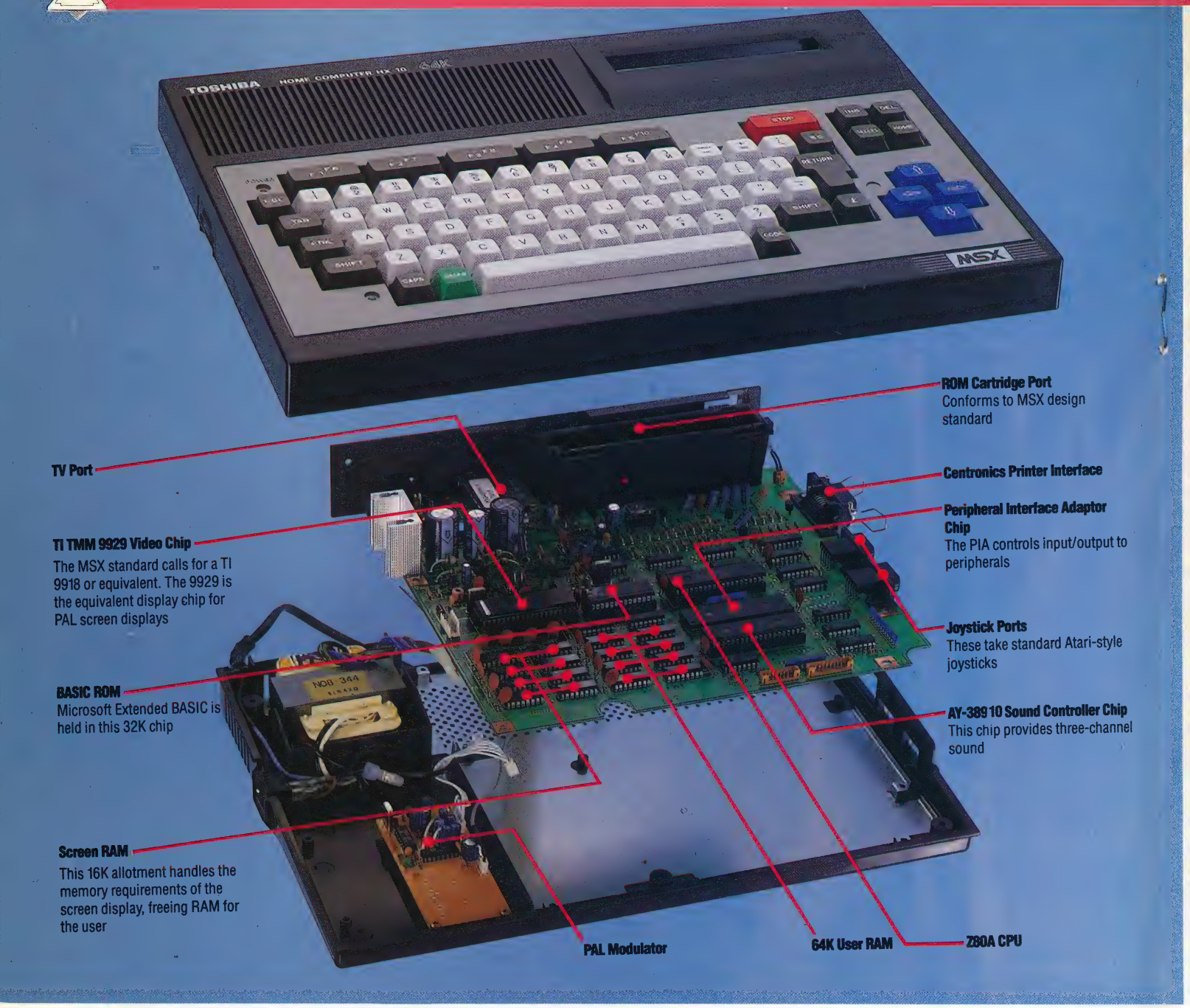
Originally, it was expected that the MSX computers would be cut-price machines, but currency fluctuations and increased manufacturing costs have pushed prices up. The Sony, for example, sells for around £300, and the Toshiba is priced at about £280. Another reason for the price increase has been the rush to get the machines onto shelves in Europe. Toshiba is sending all its machines out via costly air-freight, rather than by ship. The company has had to switch all its production lines from building Japanese versions of the machine to constructing the UK model, in the hope of being the first MSX manufacturer to have a product on sale in the UK.

One of the first things you notice when you power up an MSX micro is a row of words across the bottom of the screen. These are keywords from the BASIC language, such as RUN, CLOAD", LIST, etc. The micros have five function keys that produce these commonly used words. The words on the screen serve as labels for the function keys so that the user does not have to remember the function of each key.

These keys are automatically defined when the machines are turned on, but it is easy to change their definitions by using the KEY command. Although there are five keys only, up to 10 functions can be accessed by pressing the Shift key and the desired function key at the same time. As Shift is pressed, the labels on the screen change to

CHRIS STEVENS

Standard System
The Toshiba HX-10 has two joystick ports, a Centronics parallel printer interface, ROM cartridge port, and cursor cluster, as shown here. MSX BASIC deals with joystick control in the same way as control via the cursor, so games can be written for one type of control, and automatically make use of the other as well



reflect the new functions assigned to the keys. Each function label can contain up to 15 characters, although only the first seven of these will appear on the screen.

The keyboard and screen editor work together to make editing easy. Four keys move the cursor around the screen, and changes can be made anywhere on the screen by simply over-writing the existing characters. Inserting and deleting characters requires single keypresses. The cursor keys on the Toshiba HX-10 are the same size as other keys on the keyboard, but the Sony Hit-Bit uses large distinctive keys for its cursor cluster. These keys are used often, so such a design can be very handy.

As with the hardware, the MSX software is filled with extra features. MSX BASIC includes such commands as AUTO and RENUMber, and contains several commands to handle sound, graphics and 'interrupt handling'. There are three primary commands to create graphics. LINE draws a line between two points, although it can also be used to draw a square by adding the letter B (for 'box') after the co-ordinates. Adding the letters BF (for 'box fill') draws a square of solid colour. The CIRCLE command can be used to draw ellipses and arcs as well as basic circles. And the PAINT command will fill any outlined shape with solid colour, managing to cope with even the most awkward shapes.

Many other useful features are included in MSX BASIC, although the most impressive set of commands — for 'interrupt handling' — may not be appreciated at first. Interrupt handling is very useful in high-speed graphics programming. There are a number of situations in which a program must perform one task, while constantly checking to see if something else happens. A

Flying The Flag

MSX Standard

CPU	Z80A, 3.58 MHz
RAM	Minimum 8K
ROM	32K including BASIC
SCREEN	16 colours, 256×192 graphics, 32 sprites, 40×24 text display (or 32×24) (TI 9918 video chip or equivalent)
SOUND	3-channel, accessible from BASIC (AY38910 sound controller chip)
INTERFACES	MSX cartridge port, modulated TV output, Centronics parallel printer, cassette interface
KEYBOARD	QWERTY keyboard plus special function keys, 4 cursor keys, 10 programmable function keys

MSX Flavours

SONY HIT-BIT	Built-in database software, RGB output, optional 4K Ram packs
TOSHIBA HX-10	Expansion bus, 2 joystick ports
YAMAHA CX-5	Mini-music keyboard and software with MIDI
PIONEER	Video disk controller interface
SANYO MPC100	Optional light pen and software
JVC HC7GB	RGB output
SPECTRAVIDEO SVI 728	Full numeric keypad

Although the MSX standard calls for a minimum 8K of memory, all the above manufacturers have supplied 64K user RAM, plus 16K video RAM in their machines.

typical example of this can be found in Space Invader type games. The program must keep the aliens moving around the screen, all the while checking whether the 'fire' button has been pressed. The program needs to do two things at once, by switching rapidly between tasks.

The MSX solution is to designate certain things as 'events'. Instructions are provided to tell the computer to look out for an event. When one occurs, the computer automatically switches to a subroutine to deal with the event.

The MSX graphics screen can display 16 colours with a resolution of 256 by 192 pixels. Up to 32 eight by eight pixel sprites can be defined (or 16 sprites of 16 by 16 dots, or eight sprites of 32 by 32 dots). To make the most of the sprites, MSX BASIC includes a full set of dedicated commands, such as SPRITE to define a sprite, and PUT SPRITE to position one anywhere on the screen.

As the MSX manufacturers have claimed, plenty of cartridge software is already available for the machines. And the promise of compatibility appears to be true — software for the Toshiba HX-10 works perfectly on the Sony Hit-Bit, and vice versa. This applies both to cartridge software and cassette programs. After years of non-compatible systems, it seems almost magical to take a cartridge out of one computer and use it on another. The MSX companies are relying on this feature to make a wide range of software very quickly available for all the machines.

Whether MSX will have the market impact that the Japanese are hoping for remains to be seen. With strong competition ahead from Sinclair, Commodore and Amstrad, among others, a sales struggle looms. Nevertheless, the MSX machines do live up to their manufacturers' claims. They are well-equipped, fun-to-use computers at a reasonable price.

Standard Interface

CHRIS STEVENS

TOSHIBA HX-10 MSX

PRICE
£280

DIMENSIONS
365 × 245 × 60 mm

CPU
Z80A, 3.58 MHz

MEMORY
64K RAM (28K available for BASIC) 16K video RAM, 32K ROM including BASIC

SCREEN
40 columns × 4 rows text, 256 × 192 graphics, with 16 colours and up to 32 sprites

INTERFACES
Centronics printer, TV, monitor, audio output, 2 joystick ports, cassette port, ROM cartridge slot, expansion bus

LANGUAGES AVAILABLE
Microsoft Extended BASIC

KEYBOARD
68-key typewriter style with cursor cluster, plus 5 programmable function keys

DOCUMENTATION
Set-up guide and BASIC programming reference guide. Both are well done, but not sufficiently thorough

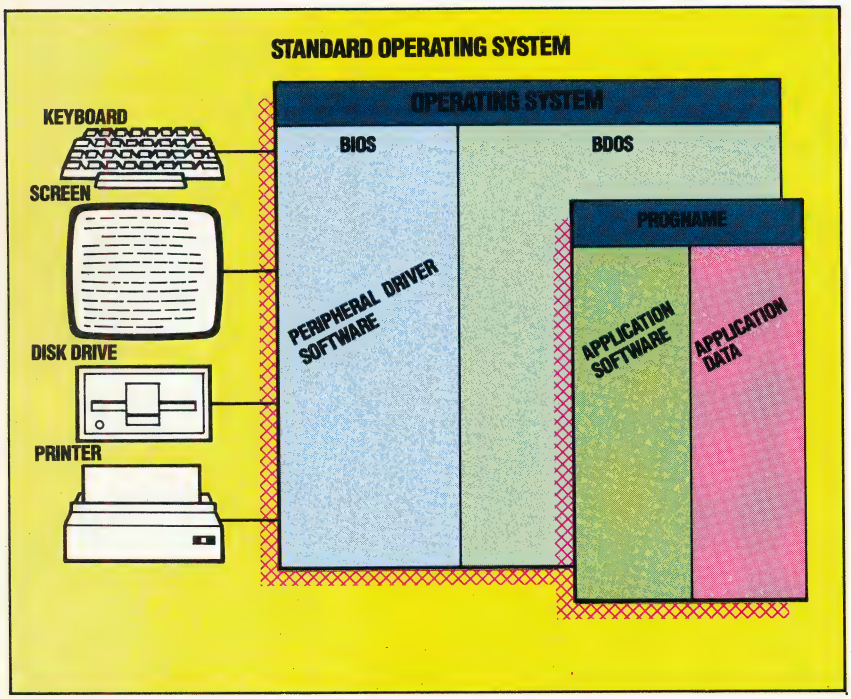
STRENGTHS
MSX BASIC has many useful features, including good graphics and sound commands. The standardisation of MSX is a strength because it will mean more software and peripherals

WEAKNESSES
MSX BASIC lacks structured programming abilities; availability of MSX and peripherals is slow in coming

DIFFERENCES
The Sony Hit-Bit has 3 programs built into ROM, RGB monitor output, and the keyboard arrangement is slightly different

COMPLETE CONTROL

In the last instalment of this series on integrated software we looked at the most common approach, that of producing all-in-one programs covering all the functions that you need. However, this is not the best system, as such programs are huge and wasteful of memory. Now we look at a more versatile method.



Operating Under Orders
With the traditional operating system the program currently running has complete charge. Its logic determines what appears on the screen, when the disk drive is to be accessed and how to interrogate the keyboard. Its general instructions are passed to the operating system, which manages the detailed driving of the particular hardware in use. The program's execution is paramount, and the operating system's subordination is taken for granted

The alternative approach to integrated software is based on a completely different principle. This relies on the computer's operating system to provide the basic facilities of integration, and individual programs written to work with that system will automatically fit and work together.

Creating such an operating system has been no easy task, since it requires the computer's hardware and software to be more sophisticated than in traditional designs. Apple has led the field with its custom-designed Lisa and Macintosh computers, although several other companies, notably Microsoft, are preparing systems to run on other popular computers such as the IBM PC.

Programs for these new operating systems are very different from programs for traditional systems. A large part of most programs deals with the user interface – the routines that receive commands and information from the user and present the results. Opinions differ on how programs should be operated, so nearly every package has its own unique operating procedures and needs to be learnt from scratch.

An integrated operating system provides a built-in set of user interface routines for every application program to use. When the program wants to display a list of options on the screen for the user to choose from, there's a ready-made routine to do it in the operating system. The advantage of this, of course, is that all the programs written to work with the operating system will have roughly the same operating procedures. Once you've learnt one program on the system, you're well on the way to using all of the others available!

One particular user interface provided for these programs is the mouse. This is a pointing device used to choose options from the screen via a corresponding cursor. An alternative is the 'touch-screen', in which a matrix of light beams responds to the touch of a finger. The display is divided into separate 'windows', each containing a different option or task. Technically, such a user interface demands a fast processor, lots of memory and very high-resolution graphics. But it is worth these extra costs because the system is generally applicable to almost any program available, it is very easy to learn and it provides the simplest possible way for the user to be able to see and switch between several applications at a time.

OPERATION CONTROL

It is important to appreciate the way this system integrates programs. The program and user are never in direct contact – everything has to be done through the operating system and the operating system is in control the whole time. In effect, each application program becomes an extension of the operating system, and the computer is a single integrated 'environment'.

This brings us onto the second major difference in the way such systems function. In a traditional system, communication between program and operating system is very much one-way. The program asks for a specific task to be carried out and the operating system subsequently does it.

In an integrated system, the operating system is in control and make demands of the program. For example, the operating system may send a message to the program that says 'Could you redraw your display because the user has just moved it to the other side of the screen' or 'Hold everything, the user has moved the mouse to a different application' or 'Here's some data for you taken from a spreadsheet.' In other words, the program has to be able to respond to the requests and demands of the operating system as well as the other way round.

Once you have this degree of co-operation between all the software on a machine, it is easy to build an integrated environment. Each program has its own window on the screen. When the user puts the mouse inside the window and chooses an option, the operating system notifies that particular program and the relevant operation is carried out.

For example, if the user moves to the corner of the window and selects the option to pick up that window and move it to a new position, routines in the operating system carry out the task and then, if necessary, inform the program of the changes so that it can amend its display appropriately. If the user takes the mouse to a different window, the original program is temporarily dormant and the operating system starts working with the new program – switching between applications is as simple as moving the mouse.

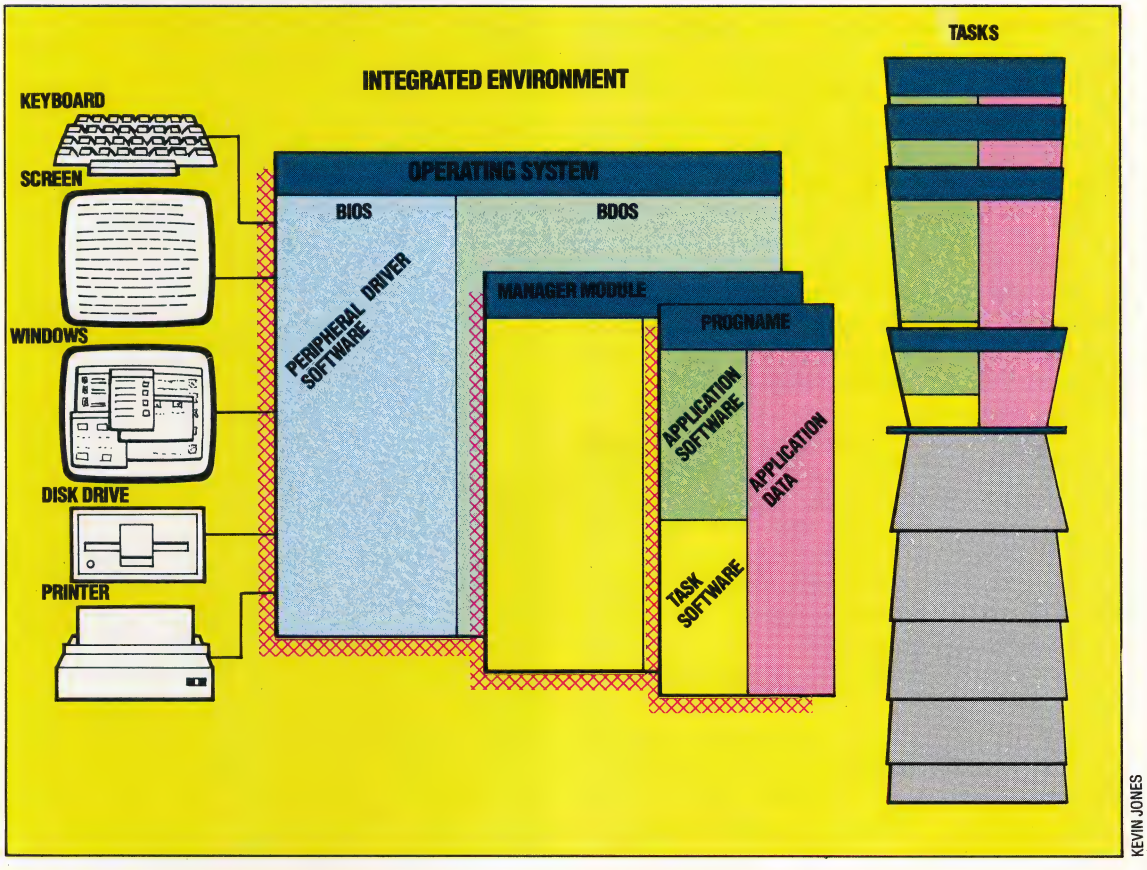
Like large all-in-one programs, such systems suggest that all the programs and information on the screen at any one time are in memory and available for use. To facilitate this, many systems have massive memories – one Megabyte on the Apple Lisa, for example, and 512 Kbytes on the Macintosh. Even then, it is usually necessary for the operating system occasionally to swap information and programs on and off disks to accommodate everything. To make the system acceptably fast, it is generally necessary for it to operate on a hard disk.

In order for data to be exchanged easily between programs, the operating system has a built-in set of formats and routines for transferring data. When you 'export' some data from one program and ask to 'import' it to another, the operating system will suspend the first program and start the second, then ask the current application to read in and process information coming from another program. These pathways can be set up automatically so that when you change information in a spreadsheet, for example, a graph of the same spreadsheet will automatically change also. The two programs don't run at the same time – the operating system merely juggles between the two of them as it needs to.

A slightly more sophisticated concept is demonstrated by Apple's Lisa, where information can be 'cut' to a clipboard window from any program and then 'pasted' into any other. Formatting information is carried with the data so that a graph produced with the business graphics software will be transferred as a graph into another program.

This then is the most sensible way to create integrated software. It enables you to mix and match any programs on the system, switch between them and move information between them easily. The drawback is that it requires sophisticated hardware that for the moment is quite expensive, and there is very little software available for you to integrate.

However, any technological innovation of this scale will take time to become commonplace. The mouse and windows interface was, for example, developed by Xerox's research teams over 10 years ago but it's taken until now for such a system to appear in the shops!



KEVIN JONES

Combined Operations
In an integrated system the operating system is enhanced by the addition of a manager module, which treats all current programs and data as 'tasks' to be scheduled and processed, and handles the underlying detailed operating system as simple system support software. This module moves tasks in and out of main memory and on and off disk according to user's requests and current task's needs. It is equipped to pass information to and from applications in standard forms, and so enables the transfer of data among the tasks. In effect the manager is a high-priority task itself, and its relationship with the other tasks is symbiotic rather than servile

RAISE THE ALARM

In the last section of Workshop, we designed and built a mains relay box that allows us to control mains voltage devices with suitable software. In this instalment, we look at the design of some programs to use the mains relay box and demonstrate some simple domestic applications.

The mains relay box is designed to control a mains supply to any device that is plugged into it. In response to a low voltage signal, the box makes or breaks the mains power fed to the socket mounted on the box. The mode of operation is such that the mains supply to the socket is maintained while there is a low voltage current supplied to the relay. Therefore, we can trigger the relay directly from the low voltage output box we built earlier in the course (see page 574). The supply of mains power from the relay box will mirror exactly the low voltage current supplied to the relay from the low voltage output box. Thus, the control of mains supply can be achieved by the same software techniques used to control low voltage devices.

If, for example, the mains relay's low voltage leads are connected to the positive and negative connections of line 0 on the output box, and it is plugged into a mains socket, then a mains current will be supplied to the socket on the relay box when bit 0 of the user port data register is sent high. Whenever bit 0 is sent low then the mains supply to the relay box's socket will stop. Up to four mains relay boxes can be connected to the low voltage output box and switched in this way.

We can make use of this simple switching arrangement to develop a number of control systems that make use of everyday household appliances. First, let's try the following simple project, in which we make use of a tape recorder to program your micro to respond 'verbally' to pressure on a pad.

To begin, we need to record a series of phrases, such as 'You're treading on my pad', followed by 'You've just done it again' and 'Look, I'm warning you!', and so on. Once the messages have been recorded, we will connect the pad and recorder to our user port system and write some software to trigger off the phrases, one at a time, in response to repeated pressure on the pad.

We have to make the following connections to the user port system:

1) Plug the mains relay voltage leads into the positive and negative terminals of line 0 on the low voltage output box.
2) Plug the supply lead to the mains relay box into a wall socket and switch it on.
3) Connect the two pressure pad leads across the positive and negative terminals of line 7 of the buffer box.

The main problem in designing the software to run this system is ensuring that the tape recorder is switched on and off with precision when a message is played. Before we can write a program therefore we must accurately time each message and enter this data into the controlling program. Timing can be done using your micro's internal timer or a stopwatch. If there are three phrases on the tape lasting for periods of T(1), T(2) and T(3) seconds then we can write a program which, on activation from the pressure pad, turns on the tape recorder for the correct time period for each successive message. If the timing of the phrases is done accurately then each phrase should be ready to start when the tape recorder is switched on.

The following programs (for the Commodore 64 and the BBC Micro) will turn the tape recorder on for three successive time intervals – T(1), T(2) and T(3) – in response to triggers from the pressure pad. You must set these variables to your own timed values for the tape you record.

BBC Micro

```
 10 REM BBC TREADING PROGRAM
 20 DIM T(3)
 30 DDR=&FE62:DATREG=&FE60
 40 ?DDR=127:REM L7 INPUT
 50 ?DATREG=0:REM ALL OFF
 60 CLS
 70 FORI=1TO3
 80 INPUT"TIME INTERVAL (SECS)";T(I)
 90 NEXT I
100 :
110 FORL=1TO3
120 CLS
130 REPEAT
140 UNTIL(?DATREG AND 128)=0:REM L7 LOW
150 ?DATREG=1:REM TURN ON TAPE
160 TIME=0:REM START TIMER
170 REPEAT
180 UNTIL TIME>T(L)*100
190 ?DATREG=0:REM TURN TAPE OFF
200 NEXT L
210 END
```

Commodore 64

```
10 REM CBM 64 TREADING PROGRAM
20 DD=56579:DATREG=56577
30 POKEDDR,127:REM L7 INPUT
40 POKEDATREG,0:REM ALL OFF
50 PRINTCHR$(147):REM CLEAR SCREEN
60 FORI=1 TO 3
70 INPUT"TIME INTERVAL (SECS)";T(I)
80 NEXT I
90 :
100 FOR L=1TO3
110 IF(PEEK(DATREG)AND128)<>0 THEN 110
115 POKEDATREG,1:REM TURN TAPE ON
120 T=TI:REM INITIALISE TIMER
130 IF T(L)>(TI-T)/60 THEN130
140 POKEDATREG,0:REM TURN TAPE OFF
150 NEXT L
160 END
```

PROGRAMMABLE ALARM CLOCK

Having developed a system sensitive to intrusive footsteps, let's now consider a project to turn your micro into a handy programmable alarm clock. Such a system can, of course, be tailored to meet one's exact needs. The program we give (in versions for the Commodore 64 and the BBC Micro) allows the user to enter:

1) the time of day;

2) the number of 'snooze' intervals (periods between the bursts of the buzzer or music) required;

3) for each snooze interval, whether music, alarm or a silent period is required and the interval length;

4) for each interval, whether a light should be switched on;

5) the latest desirable rising time.

The program works on the assumption that the following connections are made to the low voltage output box:

1) A tape recorder is connected to line 0 through a mains relay.

2) A table lamp is connected to line 1 through a mains relay to line 1.

3) A nine-volt electric bell is connected directly to line 3.

The program accepts the latest rising time and works backwards through the programmed intervals to calculate the start time for each interval. Use is made of arrays to store the data that tell us which appliances are to be on during any one period. Note that the array variables are given values that correspond to the bit value required in the data register to turn that particular appliance on. By making use of the logical OR instruction, we can simply find the composite total that must be placed in the data register to turn any combination of the devices on.

Most of our programming effort has been directed towards manipulating string variables to allow numerical calculations to be made. This is particularly true of the Commodore 64 program, as the version of BASIC used by that machine lacks the useful MOD and DIV commands available to programmers of the BBC Micro.

We have now developed a truly flexible input and output system for microcomputer control that allows us to control LEDs, low voltage devices and mains appliances, as well as allowing the micro to accept and interpret data input from a range of sensors.There are many possibilities now open to us to design control systems for our own use. In the examples given here, the micro is used as a sophisticated programmable timer. Other applications could involve turning electric fires on and off in response to a pair of heat sensors, or turning on an electric light at night. There are endless possibilities for experimentation.

IAN McKINNELL

Commodore 64

```
100 REM **** CBM 64 ALARM CLOCK ****
110 DDR=56579:DATREG=56577
120 POKE DDR,255:POKEDATREG,0
130 PRINTCHR$(147):REM CLEAR SCREEN
140 INPUT"NUMBER OF SNOOZE INTERVALS";N
150 M=N+1
160 DIM A(M),M(M),L(M),TS$(M),T(M)
170 :
180 REM **** INPUT INTERVAL DATA ****
190 FOR C=1 TO N
200 PRINT:PRINT"INTERVAL NUMBER";C
210 INPUT"MUSIC,ALARM OR SILENCE (M/A/S)";AN$
215 AN$=LEFT$(AN$,1)
220 IF AN$<>"M"ANDAN$<>"A"ANDAN$<>"S"THEN 210
230 IF AN$="M" THEN M(C)=1:A(C)=0
240 IF AN$="A" THEN A(C)=8:M(C)=0
250 IF AN$="S" THEN A(C)=0:M(C)=0
260 INPUT"LIGHT ON (Y/N)";AL$
270 L$=LEFT$(AL$,1)
280 IF L$<>"Y" AND L$<>"N" THEN 260
290 IF L$="Y" THEN L(C)=2:GOTO310
300 L(C)=0
310 INPUT"TIME INTERVAL (MINS)";T(C)
320 NEXT C
330 :
340 INPUT"LATEST RISING TIME (HHMM)";LT$
350 LT$=LT$+"00":REM ADD SECONDS
360 TS$(N+1)=LT$:REM LAST TIME
370 REM CONVERT LATEST TIME TO MINUTES
380 LM=60*VAL(LEFT$(LT$,2))+VAL(MID$(LT$,3,2))
390 :
400 INPUT"TIME NOW (HHMM)";TN$
410 TI$=TN$+"00":REM START TIMER
420 :
430 REM **** ANALYSE AND CALCULATE ****
440 REM ** CALC INTERVAL START TIMES **
450 FOR C=N TO 1 STEP -1
460 LM=LM-T(C):REM START TIME IN MINS
470 HR=INT(LM/60)
480 MN=INT(60*(LM/60-HR+.000001))
490 HR$=STR$(HR):REM HOURS
500 MN$=STR$(MN):REM MINS
510 MN$=MID$(MN$,2,LEN(MN$))
520 HR$=MID$(HR$,2,LEN(HR$))
530 REM ** ADD LEADING ZEROS **
540 SP$="00"
550 HR$=LEFT$(SP$,2-LEN(HR$))+HR$
560 MN$=LEFT$(SP$,2-LEN(MN$))+MN$
570 TS$(C)=HR$+MN$+"00"
580 NEXT C
590 :
600 REM **** GO ****
610 PRINTCHR$(147)
620 FOR C=1 TO N+1
630 IF TI$<TS$(C)THENGOSUB710:GOTO630
640 DN=M(C) OR A(C) OR L(C):REM DATREG DATA
650 POKE DATREG,DN
670 NEXT C
680 POKE DATREG,0
690 END
700 :
710 REM **** DISPLAY TIMER B/R ****
720 PRINTCHR$(145);:REM CRSR UP
730 PRINTLEFT$(TI$,2)":"MID$(TI$,3,2);
740 PRINT" "RIGHT$(TI$,2)
750 RETURN
```

BBC Micro

```
10 REM BBC ALARM CLOCK
15 MODE7
20 DDR=&FE62:DATREG=&FE60
30 CLS
40 INPUT"NUMBER OF SNOOZE INTERVALS";N
45 M=N+1
50 DIM A(M),M(M),L(M),T(M),TS(M)
70 REM **** INPUT INTERVAL DATA ****
80 FORC=1 TO N
90   PRINT"INTERVAL NUMBER";C
96   REPEAT
100     PRINT "MUSIC,ALARM OR SILENCE";
103     INPUT " (M/A/S)";AN$
105   AN$=LEFT$(AN$,1)
110   UNTIL AN$="M"ORAN$="A"ORAN$="S"
120   IF AN$="M"THEN M(C)=1:A(C)=0
130   IF AN$="A"THEN M(C)=0:A(C)=8
140   IF AN$="S"THEN M(C)=0:A(C)=0
150   REPEAT
160     INPUT"LIGHT ON (Y/N)";AL$
170     AL$=LEFT$(AL$,1)
180     UNTIL AL$="Y" OR AL$="N"
190   IF AL$="Y" THEN L(C)=2 ELSE L(C)=0
200   INPUT"TIME INTERVAL (MINS)";T(C)
210   NEXT C
220 :
230 INPUT"LATEST RISING TIME (HHMM)";LT$
232  TS(N+1)=6000*(60*VAL(LEFT$(LT$,2)))
234   TS(N+1)=TS(N+1)+VAL(RIGHT$(LT$,2))
236 REM CONVERT LATEST TIME TO MINS
237 LM=60*VAL(LEFT$(LT$,2))
239  LM=LM+VAL(RIGHT$(LT$,2))
240 INPUT"TIME NOW (HHMM)";TN$
250  TIME=6000*(60*VAL(LEFT$(TN$,2)))
255   TIME=TIME+VAL(RIGHT$(TN$,2))
260 :
270 REM ANALYSE AND CALCULATE
280 FORC=N TO 1 STEP -1
290   LM=LM-T(C):REM INTERVAL START
300   TS(C)=6000*LM
310    NEXT C
320 :
330  REM **** GO ****
340 CLS
350 FOR C=1 TO N+1
360   REPEAT
370     PROCtimer
380     UNTIL TIME)=TS(C)
390   REGDATA=M(C) OR A(C) OR L(C)
400   ?DATREG=REGDATA
420   NEXT C
430 ?DATREG=0
440 END
999 :
1000 DEF PROCtimer
1020 MIN=(TIME DIV 6000) MOD 60
1030  HR=(TIME DIV 6000) MOD 60
1040 MIN$=STR$(MIN):HR$=STR$(HR)
1042 REM ADD LEADING ZEROS
1043 SP$="00"
1044 HR$=LEFT$(SP$,2-LEN(HR$))+HR$
1045 MIN$=LEFT$(SP$,2-LEN(MIN$))+MIN$
1050 PRINTTAB(18,12)HR$;":";MIN$
1060 ENDPROC
```

FREQUENCY

At its simplest, the *frequency* of any quantity is the number of times it occurs during an observation period. The word is used most often and with a more specific meaning in physics and electronics, and in statistical analysis. In the former fields, frequency is understood to mean the number of complete cycles of vibration of some observed variable in unit time. It is measured in cps (cycles per second), but the international unit is now called the hertz. The British domestic electricity supply is an alternating current whose frequency is 50 Hz; the musical note A above middle C has a frequency of 440 Hz; the Z80A microprocessor has a clock rate of 4.25 MHz (it performs 4,250,000 primitive machine operations per second); and the BBC's Radio Four broadcasts at a frequency of 94 MHz. The frequency of vibration of wave phenomena such as sound and light is associated with two other quantities – wavelength (the distance between identical points on two consecutive cycles such as peak to peak, for example) and speed of propagation – in the simple relationship:

Speed = Frequency × Wavelength

Thus, since electromagnetic radiation (of which visible light, radio and infrared are examples) has an observed speed of propagation of 324,000 km per second, the wavelength of the Radio Four transmissions must be 3.44 metres.

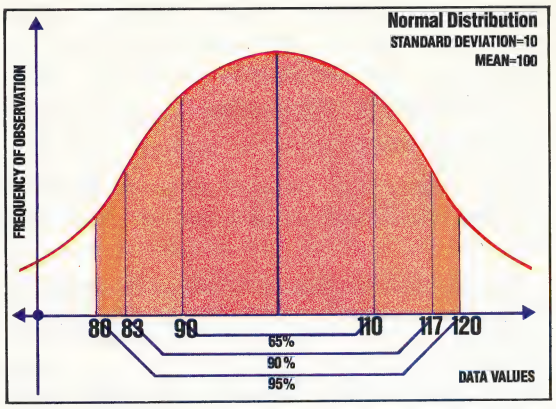
In statistics, frequency refers to the number of occurrences of some quantity in a sample or a population of similar quantities; so, in a sample of 100 British males we might observe that the frequency of occurrence of right-handedness was 87, meaning that 87 people in that sample said that they were right-handed. Statistical analysis depends upon observations of frequency of occurrence, and really derives from the speculations of French scientist Blaise Pascal (1623-62) on the expected frequency of occurrence of the numbers obtained by throwing two dice.

Give Us A Bell
The commonest frequency distribution – in nature and in theory – is the Gaussian or normal distribution, sometimes known as the bell curve

FREQUENCY DISTRIBUTION

If the results of an experiment are plotted on a graph with the results along the x-axis and their frequency of occurrence along the y-axis then the result gives a picture of the *frequency distribution* of those observations. In order to obtain a meaningful distribution it is usually necessary to group the data into sub-classes, and plot the frequency against the sub-classes; if the data group was people's heights in metres, for example, then the sub-classes might be 1.60-1.65 metres, 1.70-1.75 metres, 1.80-1.85 metres, and so on. These distributions can be analysed mathematically, and allow a great deal of descriptive and predictive information to be inferred from what would otherwise be raw data.

A commonly occurring frequency distribution is the *normal* or *Gaussian* distribution: many human attributes are *normally* distributed through the population, such as height, eye colour and (allegedly) intelligence. Many sampling distributions tend to a normal distribution as the sample size tends to infinity.



Any particular sample is characterised by its *mean* (the 'average' value of the sample data) and its *standard deviation* (a measure of the extent to which the sample data differs from the mean value). If the distribution is normal, then approximately 65 per cent of the sample data will differ from the mean value by less than one standard deviation, and over 90 per cent will differ from it by less than two and a half standard deviations. The entire science of statistics is built on this kind of analysis of frequency distributions.

FULL DUPLEX

A telephone line is *full duplex* while a radio link is usually *half duplex*: in the first case data can travel in both directions simultaneously, while in the second case the data travels in only one direction at a time – hence the need to switch from transmit to receive. An even lower level of connection is *simplex* in which data travels in one direction only, with no possibility of reversing the polarity – broadcast radio or television, for example, is a simplex communication.

FUZZY THEORY

In digital systems there are no half-measures, no uncertainty – everything is one or zero, yes or no. This binary logic is necessary at electronic levels, but it has influenced the symbolic logic with which computer programs model the real world. In this, computer thought has departed significantly from those aspects of human thought and logic that are most valuable to us, namely our ability to deal with half-truths and uncertainty, the ability to make decisions on the basis of incomplete data. *Fuzzy logic* attempts to introduce this ability to the computer by constructing a multi-value logic in which a statement may be true, probably true, possibly true, probably untrue, or untrue. This leads to some interesting insights and has so far thrown up such bizarre artefacts as fuzzy sets and fuzzy relationships. As artificial intelligence and research progresses we may expect more developments in this fascinating field.

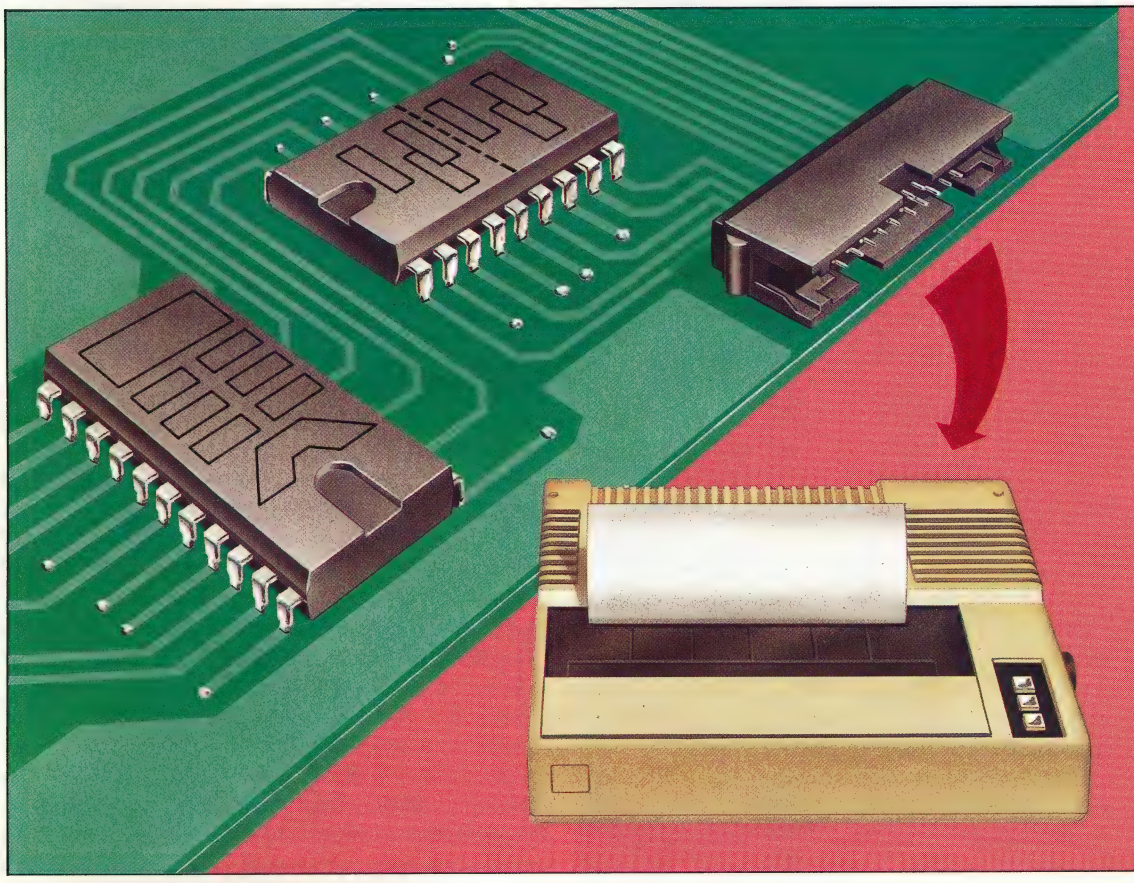
INS AND OUTS

One of the most important aspects of Assembly language programming is controlling input and output. We look at the operation of the two interface chips most commonly used with the 6809 processor — the 6820 PIA and 6850 ACIA — and show how these are programmed.

The 6809 processor, like the 6502 but unlike the Z80, does not have a separate input/output address space and special I/O instructions. Instead, the I/O device interface chips sit in the normal address space and are handled using memory access instructions. To the processor, these devices appear as memory locations exactly like the rest of memory. This has the advantage of being simple and quick, but the disadvantage of taking up a block of addresses that are then unavailable for normal use. As a consequence, the 6809, despite having a 16-bit address bus capable of addressing 64 Kbytes of memory directly, is restricted to about 56 Kbytes maximum without memory management hardware and software.

It is possible for some input/output devices to be attached directly to the system data bus but normally there is an interface chip in between. These interface chips are sophisticated devices, as complex as the microprocessor itself, and it is normal to use chips belonging to the same family as the processor since this makes the job of attaching them and controlling them easier. The two chips most commonly used with the 6809 are the 6802 (or 6821) PIA (peripheral interface adaptor), which handles parallel I/O, and the 6850 ACIA (asynchronous communications interface adaptor), which deals with serial I/O. Each of these has a number of registers, and controlling them is a matter of reading and writing the contents of those registers, treating them as though they were normal memory locations. There are three types of registers:

- *Control Registers:* These are write-only registers; values are stored in them in order to program the chip for the particular options that you require, such as setting the baud rate.
- *Status Registers:* These are read-only registers, the values of which give an indication of the 'status' of the chip. These will show, for example, whether an input has been received, whether the last output has been transmitted, or whether an error has occurred.



Peripheral Matters

Printers require data to be sent to them in particular formats and at certain speeds: it would be wasteful to have the CPU deal with such relatively trivial matters, so the CPU sends the character data to the Peripheral Interface Adaptor, and it devotes itself full-time to communicating with the printer

KEVIN JONES

● *Data Registers:* These are the registers that contain the data being input or output, and so they may be read-write or separate read and write.

In order to conserve memory space it is common practice for more than one register to occupy the same address. For example, a status register and a control register may be at the same address; the one that appears at that address is determined by whether you are reading or writing to it. Similarly, an input data register and an output data register may share an address.

The 6820 PIA contains six registers and occupies four consecutive bytes of memory space. The chip actually contains two independent ports, each of which uses three registers. The peripheral side of the chip has eight data lines and two control lines for each port. The two control lines would be connected to appropriate control lines on the peripheral so that they can be used to determine status. Control line 1 is for incoming control signals only, but control line 2 can be programmed to receive or send control signals.

The three registers are:

● A data register, which can function for both input and output, since each bit can be independently set.

● A data direction register, each bit of which is used to set the corresponding bit in the data register as input (0) or output (1).

● A combined control/status register.

The data direction register and the data register share the same address. The state of one of the bits in the control register determines which of these appears at this address. The table in the margin gives the offset from the base address of the chip for the addresses of each of the registers.

Register	Offset
Data A	0
Data Direction A	0
Control/Status A	1
Data B	2
Data Direction B	2
Control/Status B	3

The bits in the control/status register are assigned as follows:

Bit	Function
7	Status bit for control line 1; set to one when a control signal is received and automatically cleared to zero when the data register is read
6	Status bit for control line 2; as bit 7
5	Determines whether control line 2 is used for input (zero) or output (one)
4	Determines the nature of the control signal on line 2
3	If control line 2 is set for input then a one here enables interrupts from bit 6; if output then it helps determine the nature of the signal
2	Selects between data (1) and data direction (0) registers
1	Determines the nature of the control signal on line 1
0	A one here enables interrupts from bit 7

For the moment we shall not be considering the use of interrupts, nor shall we be concerned with the detailed effects of bits 1 and 4. Note that when writing to the register to set the control bits it is impossible to affect bits 6 and 7.

The first of our example programs sets up and uses a 6820 chip to control a printer via a standard Centronics interface. The latter specifies a large number of control lines as well as the eight data lines. We are not concerned with the detail of these except to note that one control line (called the *strobe*) is used to signal to the printer that a character is on the way. This should be connected to control line 2, which must be set for output. Another control signal (termed the *acknowledge*) is used by the printer to indicate that it is ready for the next character to be sent. This should be connected to control line 1. The eight data lines should clearly be connected to the eight data outputs from the PIA port.

To set up the port we must select the data direction register and program all eight bits for output, then select the data register, and set control line 2 for output. To use the chip we continually read the control/status register until a one appears in bit 7, indicating that the printer is ready for a character. We can then write a character to the data register, which automatically sends a control signal out on control line 2. Bit 6 will be set to one when the character has been transmitted. We must then read the data register to clear bits 6 and 7 and repeat the process until the last character has been transmitted. The process of sending and receiving control signals between the processor and the peripheral is known as *handshaking*.

We shall assume that the base address of the PIA is given in a table of addresses located at $3000. On entry to the printing subroutine, processor register A contains the index into this table, and Y contains the address of the string to be printed. The string is stored in the normal format; that is, length byte first. There are two subroutines, one to set up the port and one to print the string.

6850 ACIA

The 6850 ACIA is a UART (universal asynchronous receiver/transmitter) that is used for serial communication, normally using the RS232 protocol and possibly a modem. It has four registers and occupies two addresses. There are five connections to the chip on the peripheral side: one line is for transmitted data, one is for received data, and three control lines are for handshaking, if this is required. Two of these are for incoming control signals — DCD (Data Carrier Detect) and CTS (Clear To Send) — and one is for outgoing signals — RTS (Request To Send). The uses of these lines should be fairly obvious from their names, and they may be connected to the similarly named lines on a standard RS232.

The four ACIA registers are given in the margin. In the control register, the most significant bit (bit 7) is used to enable interrupts for receiving data. Bits 5 and 6 are used to enable or disable transmission interrupts and to determine the nature of the control signal sent out on the RTS line. Bits 2, 3 and 4 are used to determine the size of the 'package' that is actually transmitted. When a byte is transmitted over a serial link there are usually at least 10 bits sent, beginning with a start bit, which is detected by the receiver so that it knows that data is following. The actual data itself

Register	Offset
Control Register	0
Status Register	0
Transmit Data	1
Receive Data	2

can be seven or eight bits, and there may be a parity bit appended to that data. A parity bit is an extra bit that helps detect transmission errors. Finally, there may be one or two stop bits. The various options available are as follows:

Control Register			Number Of Data Bits	Parity	Number Of Stop Bits
Bit 4	Bit 3	Bit 2			
0	0	0	7	Even	2
0	0	1	7	Odd	2
0	1	0	7	Even	1
0	1	1	7	Odd	1
1	0	0	8	None	2
1	0	1	8	None	1
1	1	0	8	Even	1
1	1	1	8	Odd	1

The two least significant bits (0 and 1) are used to determine the speed of transmission and reception. This is done by setting a divisor for the clock rate. The 6850 does not have its own clock, and therefore must be provided with an external one, typically set at 1,760 Hz.

Control Register		Clock Rate Divisor
Bit 1	Bit 0	
0	0	1
0	1	16
1	0	64

The combination not shown in the table, when both these bits are one, causes a master reset of the chip.

In the status register the bits have the following functions:

Bit	Function
7	Interrupt request
6	Set if a parity error occurs on reception
5	Set if the receiver overruns, i.e. too many bits were received, with characters running into the previous ones
4	Set if a framing error occurs on reception — the wrong number of start and stop bits
3	Set when a signal is received on the CTS line
2	Set when a signal is received on the DCD line
1	Set when the transmit data register is empty
0	Set when the received data register is full

Our second example program uses a 6850 chip to receive a character string, terminated by a carriage return, from a remote terminal. The principle is to program the chip appropriately, then loop round checking if the receive data register is full. When it is, we remove the data byte, which resets bit 0 in the status register. The process is repeated until the character received is a carriage return (ASCII code 13). We shall be ignoring any transmission errors, though checking for them by masking the contents of the status register to see if any of the error indicating bits are set is quite straightforward. We shall assume a fairly common protocol: eight data bits, no parity and two stop bits and a divide by 16 clock speed. The first subroutine programs the chip, the second receives the data.

PIA Program

```
TABLE   EQU   $3000
        ORG   $1000
                        Subroutine to set up Port A
        ASLA            Shift A left to multiply by two
                        (the table consists of two-byte
                        addresses)
        LDX   #TABLE
        LDX   A,X       Get base address of PIA
        CLR   1,X       Get access to data direction
                        register
        LDB   #%11111111
        STB   ,X        Set all bits for output
        LDB   #%00101100
        STB   1,X       Disable interrupts, set control
                        line 2 for output and select
                        data register
        RTS
                        Subroutine to print string
                        whose address is in Y
        ASLA            Shift A left to multiply by two
        LDX   #TABLE
        LDX   A,X       Get base address of PIA
        LDA   ,Y+       Get length of string in A
LOOP1   BEQ   FINISH    Check if length is zero
LOOP2   LDB   1,X       Check if ready for next bit
        ANDB  #%10000000
                        Mask off all except bit 7
        BEQ   LOOP2     If not ready
        LDB   ,Y+       Get next character
        STB   ,X        Print it
LOOP3   LDB   1,X       Check if transmitted
        ANDB  #%01000000
                        Look at bit 6
        BEQ   LOOP3     Loop if not ready yet
        LDB   ,X        Read data register to clear
                        status bits
        DECA            Subtract one from length
        BRA   LOOP1     Get next character
FINISH  RTS
```

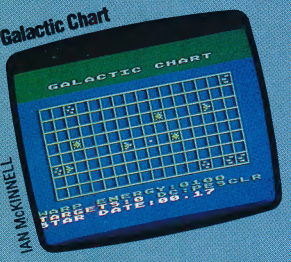
ACIA Program

```
TABLE   EQU   $3000
        ORG   $1000
                        Subroutine to program 6850
                        Subroutine to set up ACIA
ACIAST  ASLA            Shift A left to multiply by two
                        (table of two-byte addresses)
        LDX   #TABLE
        LDX   A,X       Get base address of ACIA
        LDA   #%00000011
                        Master reset of ACIA
        STA   ,X        Into control register
        LDA   #%00010001
                        Program ACIA (8 data bits, no
                        parity, 2 stop bits)
        STA   ,X
        RTS
                        Subroutine to accept string of
                        characters
BUFFER  EQU   $4000     Somewhere to put the string
CR      EQU   13        ASCII for carriage return
        BSR   ACIAST    Set up ACIA. X register
                        contains ACIA address
        LDY   #BUFFER   Destination in Y
LOOP    LDB   ,X        Get status
        ASLB            Shifts bit 7 out of B register
                        into carry flag of CCR
        BCC   LOOP      Go back if that bit was not set,
                        i.e. there is no interrupt
                        request yet
        LDA   1,X       Get data byte
        STA   ,Y+       Store it
        CMPA  #CR       Is A a carriage return?
        BNE   LOOP      Next character
        RTS
```

STAR WARS

An Encounter With The Zylons

Galactic Chart



IAN McKINNELL

Star Raiders is a development of the popular computer game Star Trek. Unlike all the other games in this series of Atari classics it was designed specifically for home computers and offers plenty of excitement for both novice and skilful players.

In Star Raiders the player takes the role of commander of the spaceship Star Raider and travels around the galaxy in pursuit of the enemy craft, the Zylons. The game requires the use of the keyboard as well as joystick control. After moving to a sector of the galaxy, pressing the F key will display the forward view from the cockpit. The position of the Zylon ships is indicated by gauges at the bottom of the screen, and by pressing the L key the player calls up the 'long-range scanner', giving a view of the current grid sector with the player's craft in the centre and the Zylons massing in the distance.

The player must then engage the enemy – either by using the standard engines, which may well result in the Zylons escaping, or by going into 'hyperspace' (achieved by pressing H), in which case the distance will be covered in a few seconds. Before engaging hyperspace, the tracking computer must be utilised by pressing T; if this is not done, the hyperspace jump may well result in the player's craft ending up in a totally unknown sector of the galaxy. Other factors to consider are the use of the attack computer (accessed by the C key) and the defence shields of the Star Raider, which are powered by the use of the S key.

Once hyperspace has been left, the computer flashes a Red Alert warning and the battle is on. The Zylon craft attack the Star Raider from all sides, growing ever larger as they approach. Joystick control enables the player to rotate the craft in all directions, and the speed of the ship is set by use of the numeric keys. The measure of control offered enables the player to indulge in dogfights, diving and swooping after the enemy.

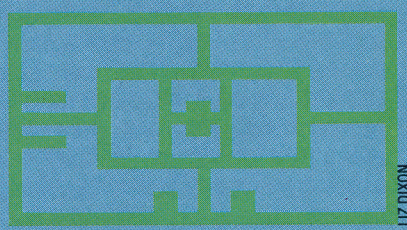
But the dogfights use up a lot of energy, and enemy craft will score numerous hits on the Star Raider, necessitating a visit to a starbase to refuel and effect repairs. The player must then move to a grid containing a star; as the Star Raider approaches, the star will grow into a large yellow flying saucer. The game's hardest manoeuvre then follows, as the player attempts to move into orbit around the base. The starbase must first be fixed in the ship's cross-hair sights, and the player must slow the ship's momentum until the target distance meter registers zero. Once this has happened, the Star Raider may be halted, and if the manoeuvre is successful, the message ORBIT ESTABLISHED will flash across the top of the screen. Great care must be taken during this approach, as it is all too easy to overshoot the target. Once orbit is established, a refuelling pod emerges from the starbase and docks with the Star Raider, enabling the ship to return to the fray.

Every so often a message reading STARBASE SURROUNDED appears onscreen, and the player must then hurry to the beleaguered station to prevent its destruction. When attempting to defend the starbase, the player must exercise great care to avoid hitting the starbase with the Star Raider's weapons.

Star Raiders offers four levels of play, ranging from 'Novice' to 'Commander'. At lower levels, the player does not need to worry too much about damage to the Star Raider, as there are few Zylons and their fire is not very accurate. At higher levels, survival for even a few minutes is extremely difficult.

Star Raiders was awarded the title of Game of the Year for 1980 in the United States and fully deserves the praise it has received. It is a pity that Atari has not made the game available for a wider range of computers.

Star Raiders: For all Atari computers, £9.99
Publishers: Atari Corporation UK Ltd, Atari House, Railway Terrace, Slough, Berks.
Authors: Atari
Joystick: Required
Format: Cartridge



LIZ DIXON

DATABASE

Here, courtesy of Zilog Inc., we produce another part of the Z80 programmers' reference card.

General-Purpose Arithmetic and CPU Control Groups

General-Purpose Arithmetic

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

Miscellaneous CPU Control

'NOP'	00
'HALT'	76
DISABLE INT '(DI)'	F3
ENABLE INT '(EI)'	FB
SET INT MODE 0 'IM 0'	ED 46
SET INT MODE 1 'IM 1'	ED 56
SET INT MODE 2 'IM 2'	ED 5E

8080A MODE

RESTART TO LOCATION 0038_H

INDIRECT CALL USING REGISTER 1 AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands.	↕	↕	X	↕	X	P	•	↕	00 100 111	27	1	1	4	Decimal adjust accumulator.
CPL	$A \leftarrow \overline{A}$	•	•	X	1	X	•	1	•	00 101 111	2F	1	1	4	Complement accumulator (one's complement).
NEG	$A \leftarrow 0 - A$	↕	↕	X	↕	X	V	1	↕	11 101 101 01 000 100	ED 44	2	2	8	Negate acc. (two's complement).
CCF	$CY \leftarrow \overline{CY}$	•	•	X	X	X	•	0	↕	00 111 111	3F	1	1	4	Complement carry flag.
SCF	CY ← 1	•	•	X	0	X	•	0	1	00 110 111	37	1	1	4	Set carry flag.
NOP	No operation	•	•	X	•	X	•	•	•	00 000 000	00	1	1	4	
HALT	CPU halted	•	•	X	•	X	•	•	•	01 110 110	76	1	1	4	
DI ★	IFF ← 0	•	•	X	•	X	•	•	•	11 110 011	F3	1	1	4	
EI ★	IFF ← 1	•	•	X	•	X	•	•	•	11 111 011	FB	1	1	4	
IM 0	Set interrupt mode 0	•	•	X	•	X	•	•	•	11 101 101 01 000 110	ED 46	2	2	8	
IM 1	Set interrupt mode 1	•	•	X	•	X	•	•	•	11 101 101 01 010 110	ED 56	2	2	8	
IM 2	Set interrupt mode 2	•	•	X	•	X	•	•	•	11 101 101 01 011 110	ED 5E	2	2	8	

NOTES: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop
★ indicates interrupts are not sampled at the end of EI or DI

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.